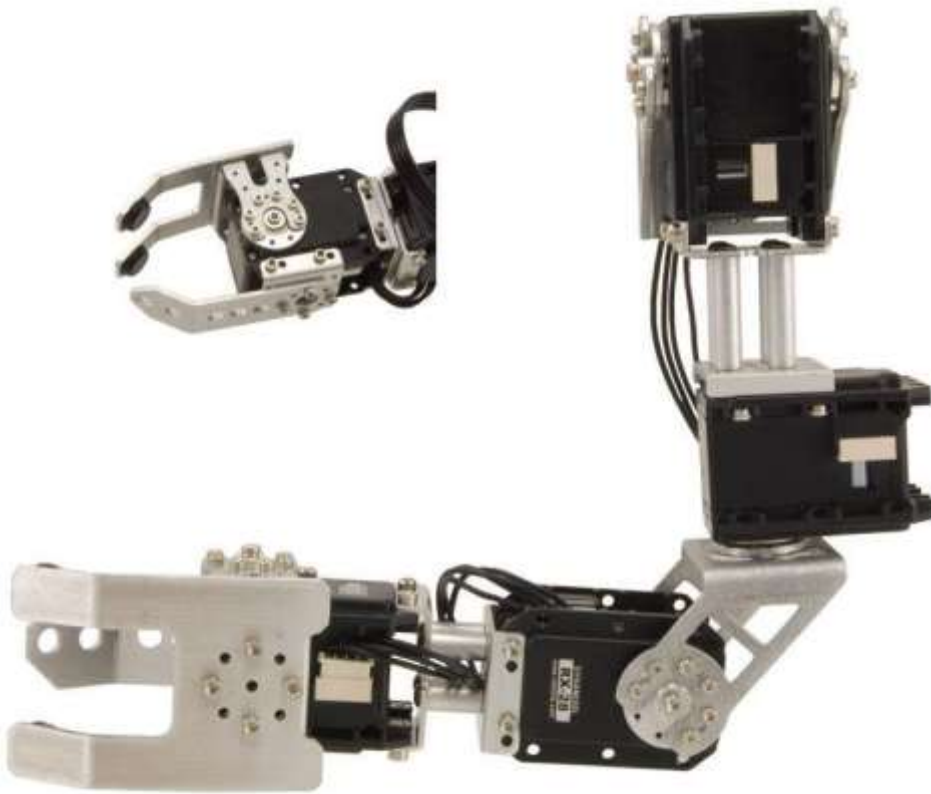


Introduction to Dynamixel Motor Control Using OpenCM

Daniel Jacobson

Alexander Hunt



Robotic Jointed Arm Using Dynamixel Motors

This is an update to the Introduction to Dynamixel Motor Control Using the Arbotix-M originally written by Daniel Jacobson in fulfillment of a Masters Project in 2015 at Case Western Reserve University. Since that time, we have migrated to using the Robotis OpenCM which is less expensive, faster, and more robust. In this update, some components may be missing information, or there may be outdated or old references that are not relevant. I also borrowed heavily from several documents written by Dr. Nicholas Szczecinski. However, I have attempted to update this document to the best of my ability to reflect methods and protocols appropriate for the OpenCM controller. If you notice any issues or have concrete recommendations for improving the document, please send them to me at ajh26@pdx.edu.

Thanks,

Dr. Alex Hunt

Portland State University

Contents

[Preface](#)

[Purpose of this Text](#)

[How to use this Text](#)

[Acknowledgements](#)

[Chapter 1: Introduction to Dynamixel](#)

[1.1 What is Dynamixel?](#)

[1.2 Dynamixel Applications](#)

[Chapter 2: Introduction to Arduino and ArbotiX](#)

[2.1 What is Arduino?](#)

[2.2 Advantages and Disadvantages of Arduino](#)

[2.3 ArbotiX-M Robocontroller](#)

[Chapter 3: Controller Setup](#)

[3.1 Software Downloads](#)

[3.2 Software Installation](#)

[3.3 Hardware Information](#)

[3.3.1 Necessary Hardware](#)

[3.3.2 Recommended Hardware](#)

[3.3.3 Additional Hardware for Specific Applications](#)

[3.4 Hardware Setup](#)

[3.4.1 Robocontroller Connections – Getting the ArbotiX-M to Blink](#)

[3.5 Setup Test – Making ArbotiX-M Blink](#)

[Chapter 4: Dynamixel Programming](#)

[4.1 Introduction to Low Level Dynamixel Programming](#)

[4.1.1 Binary Data and Decimal Conversion](#)

[4.1.2 Hexadecimal Representation](#)

[4.1.3 Dynamixel Communication Overview](#)

[4.1.4 Dynamixel Instructional Packets](#)

[4.1.5 Dynamixel Status \(Return\) Packets](#)

[4.1.6 Dynamixel Packet Examples](#)

[4.2 Programming Dynamixel Motors Using the Arduino IDE and ax12 Library](#)

[4.2.1 Brief Glossary of Terms for Arduino](#)

[4.2.2 Arduino Basics](#)

[4.2.3 Preparing the Hardware and Sketch](#)

[4.2.4 Changing the Position of a Dynamixel Motor](#)

[4.2.5 Using ax12SetRegister and ax12GetRegister](#)

[4.2.6 Using ax12SetRegister2](#)

[4.2.7 Defined Variables](#)

[4.3 Controlling Several Motors Simultaneously](#)

[4.4 Writing Additional Functions for Modular Programming](#)

[4.4.1 Simple Function Example](#)

[4.4.2 Function with Parameters and Return Value](#)

[Chapter 5: Creating and Extending Libraries](#)

[5.1 Create Your C++ File](#)

[5.1.1 #include](#)

[5.1.2 Creating Functions](#)

[5.1.3 Limiting Torque](#)

[5.1.4 Applying Torque](#)

[5.2 Create Your Header File](#)

[5.2.1 #define and #include](#)

[5.2.2 Referencing Functions](#)

[5.2.3 Ending the Header File](#)

[5.3 Using Our Library](#)

[Chapter 6: Real-time Control Using MATLAB](#)

[6.1 Serial Communication Revisited](#)

[6.2 Sending Data through Serial Connection in MATLAB](#)

[6.2.1 Defining the Problem \(or Goal\)](#)

[6.2.2 Uploading a Sketch to Receive and Interpret Data](#)

[6.2.3 Sending Serial Data over MATLAB](#)

[6.3 Serial Connection Summary](#)

[Chapter 7: Real Time Dynamixel Control Using C++](#)

[7.1 MATLAB vs. C++](#)

[7.2 The Arduino "Receiving" Sketch](#)

[7.3 C++ Windows Form Application for Serial Communication](#)

[7.3.1 Creating a C++ Windows Form Application for Serial Communication](#)

[7.3.2 Adding Buttons and Other Controls to C++ Windows Form](#)

[7.3.3 Making Windows Forms Controls Carry out Functions](#)

[7.4 Other Serial Resources](#)

[Appendix 1: Sources](#)

[Appendix 2: Additional Resources](#)

[Appendix 2.1 Manually Changing Dynamixel ID](#)

[Appendix 2.2 Restoring Dynamixel to Factory Defaults](#)

[Appendix 3: AX-12 Property Address Table](#)

[Appendix 4: Code Base for Examples in This Text](#)

[Appendix 4.1 Arduino Sketches](#)

[Appendix 4.1.1 Dyna_IDChange](#)

[Appendix 4.1.2 Dyna_PositionSet](#)

[Appendix 4.1.3 Dyna_PositionSet2](#)

[Appendix 4.1.4 DynaTorqueTesting](#)

[Appendix 4.1.5 MatlabControl](#)

[Appendix 4.1.6 FunctionTest](#)

[Appendix 4.2 C++ Library Code](#)

[Appendix 4.2.1 Header File](#)

[Appendix 4.2.2 C++ File](#)

[Appendix 4.3 MATLAB Serial Control Code](#)

[Appendix 4.3.1 Serial Object Definition](#)

[Appendix 4.3.2 positionSet Function](#)

[Appendix 4.4 C++ Serial Control Code](#)

[Appendix 4.4.1 Form Layout](#)

[Appendix 4.4.2 openSerial_Click Method](#)

[Appendix 4.4.3 closeSerial_Click Method](#)

[Appendix 4.4.4 changePosition_Click Method](#)

Preface

Purpose of this Text

This text is meant to be a practical guide to allow beginners in robotics to get their projects up and running as quickly as possible. It will guide you through the necessary software and hardware setup to control individual Dynamixel motors. It will explain the data flow and low level data analysis behind the control of Dynamixel motors to give a greater understanding of how the motors function. It will show you the basic existing commands to control the motors. It will introduce you to library creation for more complex motor commands. It will also show you basic serial communication using MATLAB to create real-time motor controllers. Along the way there will be many additional resources provided as hyperlinks that you can navigate to for further information. All of the information provided in this text is applicable to more complex Dynamixel applications; however, this text will focus mainly on controlling individual motors in simple applications. Real applications will utilize multiple motors, but the extension of topics discussed in this text is as simple as just plugging in the extra motors. A brief discussion of multiple motor control can be found in section 4.3.

This text will not teach you how to define the dynamics of your robotic applications. The scope of the text focuses mainly on hardware control; it is up to you to define your own dimensions and dynamics of your robot.

The intended audience for this text is beginners in robotics; whether hobbyists or researchers, I hope this text can jump start the practical side of your robotics projects. I highly recommend having some object oriented programming (OOP) experience before following this text. Dynamixel motors are smart servos that are controlled entirely by software commands; complex motor control depends largely on extensive libraries with useful functions (requiring OOP experience). As you will discover later in this text, it would be extremely difficult to avoid OOP and instead control the motors by sending low level data.

How to use this Text

Throughout the manual, various websites will be referenced in hyperlinks; therefore, this document is best viewed on a computer with internet connection. Hyperlinks will appear in blue and will be underlined (like [this](#)), signifying that you can click on them for further information or for a download. Several links will also be **[bolded](#)**, signifying that it is pertinent to visit the page to continue in the manual. Usually, this will only occur for a necessary download or for an item that must be purchased (or if already at your disposal, an item that must be used). If a link is not bolded, it means you can find additional information that is not pertinent to the successful completion of this text. I personally recommend skimming the majority of them as soon as you encounter any confusing topics. A lot of them give a more thorough review of the material presented.

It is also important to make sure you are following along the steps listed in the text with the hardware in front of you. It will be difficult to learn from this text without physically going through the steps listed.

Acknowledgements

I'd like to acknowledge Nick Szczecinski and David Chrzanowski, for without them this text would not exist. Their introduction into the realm of robotics has led to quite the experience, and their invaluable

assistance along the way has led to this final product. Acknowledgement is also due to Akhil Kandhari for spending considerable time talking through technical problems as they arose. Thanks is also owed to Matt Klein and Andrew Horschler for their technical tutorials in the realm of low-level hardware programming. Without their assistance, I would have been lost from the start. Finally, I'd like to acknowledge my advisor Dr. Roger Quinn for providing me with the necessary resources to be successful.

Chapter 1: Introduction to Dynamixel

1.1 What is Dynamixel?

[Dynamixel](#) is a line-up of high performance networked actuators for robotic applications. They are made by the Korean manufacturer Robotis. Each motor is fully integrated with feedback functionality and programmability. The Dynamixel line-up contains motors that can be controlled with either the [TTL](#) network interface or the [RS-485](#) network interface. The OpenCM microcontroller supports TTL interface by default; other controllers can be used for RS-485 interfacing motors. The list of which motors support each network interface can be found at the [Dynamixel](#) Wikipedia page.



Figure 1.1 - Line-up of Dynamixel Motors

1.2 Dynamixel Applications

Dynamixel motors take advantage of an all-in-one structure to provide immense versatility in robotic applications. Position feedback and control is standard in all motors, and direct torque control is available in several specific models. Multiple motors can be networked together through a daisy chain, and each motor has a modifiable ID value to easily control specific motors in the network.

Some specific applications include the [Giger Humanoid Robot](#), [Bioloid Robotic Training Kit](#), [Robotic Arms and Cameras](#), and many more. The modular nature of Dynamixel motors allows them to be used for virtually any small scale robotic application.

Chapter 2: Introduction to Arduino and OpenCM

2.1 What is Arduino?

"Arduino is an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software" (www.arduino.cc). In other words, Arduino is a low-level hardware platform with a vast collection of [libraries](#) that allow users to quickly create and implement hardware solutions. Arduino is a combination of both hardware and software that interface well together. The OpenCM microcontroller

that will be used in this text is an Arduino based microcontroller designed specifically for use with Dynamixel motors. The Arduino Integrated Development Environment (IDE) is used to write code and upload code to the OpenCM controller which will control the motors. The Arduino programming language is based on C and C++, and the language reference for the Arduino language can be found [here](#).

2.2 Advantages and Disadvantages of Arduino

The biggest advantages of using an Arduino based controller are that it is relatively easy to use, and there is a huge community of developers familiar with Arduino who can be of assistance through online means of communication such as forums. The software is proven and pretty well supported, and extending/creating libraries is relatively simple with familiarity in object oriented programming (especially C++). Another advantage is that Arduino boards are inexpensive compared to the overall price of a robot. The OpenCM controller is only \$20, while some of the Dynamixel motors cost over \$100.

The biggest disadvantage of using an Arduino based controller is power. Most Arduino boards (including the OpenCM) do not have the fastest processors or the best memory (a result of their low price). A workaround to this disadvantage is using multiple Arduino controllers and networking them together or networking them all to a single controller (like a PC).

2.3 ArbotiX-M Robocontroller

The [ArbotiX-M](#) robocontroller is an Arduino based robotic controller used for small to medium robotic applications. This controller is designed specifically with Dynamixel in mind, and it allows direct connection between the Dynamixel network and the board. It has a 16MHz processor (ATMEGA644p), two serial ports, three TTL Dynamixel network interfacing ports for direct motor connection, as well as digital and analog pins. The serial ports make it easy to program using [FTDI](#) to USB connection. This text may refer to this board as both ArbotiX and Arboitx-M (the M stands for mini; it is smaller than the original ArbotiX controller without sacrificing any of the functionality).

2.4 OpenCM Microcontroller

The Open CM9.04 C is an open source micro-controller from ROBOTIS powered by the 32-bit ARM Cortex-M3 processor. Any 3-pin DYNAMIXEL servo using the TTL communication protocol can be used with the OpenCM9.04-C. It is also designed specifically with the Dynamixel in mind. In addition to the improved processor a standard micro-USB cord can be used to power and program the OpenCM board. However, USB power cannot be used to operate the DYNAMIXEL's themselves. Separate power supply needs to be provided. (OpenCM9.04 can operate using power supplied via USB, battery, + - terminal.).

Chapter 3: Controller Setup

The two major components that need to be set up for Dynamixel motor control are the hardware and the software. This chapter will guide you through the setup step by step.

3.1 Software Downloads

Before beginning, the following software must be downloaded:

- Arduino IDE (Integrated Development Environment)
 - o May be downloaded [here](#)

If you are using a single computer for all your work, feel free to download the 'Windows Installer'. If however, you are using a shared computer, or you will be transferring the work between multiple computers, I recommend you download the 'Windows ZIP file for non admin install', and place this on a portable USB drive. If you are using a shared computer, and attempt to install Arduino using the shared computer's drive, you will find that your downloaded libraries have been eliminated and will need to re-install them each time (~20 min process).

3.2 Software Installation

When the above file has been downloaded, complete the following steps to properly install the IDE and the libraries. The figures below provide visual guidance to the installation process.

1. Install Arduino IDE in a file location (preferably on a USB drive) you will remember.
2. Open the Arduino IDE.
3. Click File>Preferences.
4. Change the "additional boards manager url" to https://raw.githubusercontent.com/ROBOTIS-GIT/OpenCM9.04/master/arduino/opencm_release/package_opencm9.04_index.json
5. Click OK.
6. Click Tools>Board>Board Manager.
7. Search for "opencm" and install the OpenCM9.04 board package. This may take 10 or 20 minutes. This will also install the DynamixelSDK library (You may have to refresh Arduino first).
8. Now, you should be able to use the DynamixelSDK and DynamixelWorkbench packages. For example, you can click File>Examples>OpenCM9.04>08. Dynamixel Workbench>j_Position, and control the position of a Dynamixel servo.

3.3 Hardware Information

The pieces of hardware that will be referenced in this document are listed below:

3.3.1 Necessary Hardware

- OpenCM9.04-C
 - May be purchased [here](#)
 - Additional hardware information about the OpenCM microcontroller can be found [here](#)
- AX-12A Dynamixel Motor
 - May be purchased [here](#)
- MX-64T Dynamixel Motor
 - May be purchased [here](#)
 - **Note: To complete this text, only the AX-12A motors are required; the MX-64T motors are much more powerful**
- AX/MX Power Hub
 - May be purchased [here](#)
- microUSB Cable
 - May be purchased [here](#)
- 12V Power Supply
 - May be purchased [here](#)
 - **Note: If using your own supply, make sure the current is rated appropriately for the motors you are using. Refer to your motor's manual to see current ratings.**
- 3 Pin Dynamixel Compatible Cables (at the length of your choice)

- o May be purchased [here](#)

3.3.2 Recommended Hardware

- Dynamixel U2D2 Adapter
 - o May be purchased [here](#)

3.3.3 Additional Hardware for Specific Applications

- RX Bridge for ArbotiX Robocontroller
 - o May be purchased [here](#)
 - o **Note: This will allow the ArbotiX robocontroller to connect to RS-485 motors, but it is NOT compatible with the ArbotiX-M. Only buy this if you have the original ArbotiX controller and need to connect to RS-485 motors.**

3.4 Hardware Setup

3.4.1 Microcontroller Connections – Getting the OpenCM to Blink an LED

The first thing to properly setup is the OpenCM microcontroller. In order to do this, plug in the USB to OpenCM input (see Figure 3.4). Plug the USB end into your computer.



Figure 3.4 – Computer to OpenCM

3.5 Setup Test – Making OpenCM Blink

The first thing we are going to program our OpenCM controller to do is to have the LED blink at defined intervals. This is the easiest and quickest way to check our software and hardware setup to make sure everything is connected and installed correctly.

With the OpenCM controller connected to the USB cable, and with the USB end connected to your computer, perform the following steps:

1. Open the Arduino IDE
2. Navigate to “Tools-> Board” and select “OpenCM9.04 Board”
3. Navigate to “Tools-> Serial Port” and select the connection that corresponds to your OpenCM microcontroller
 - a. To determine which port to choose, you can unplug the controller and navigate to “Tools-> Serial Port” and see which one disappears; plug it back in and choose the one that disappeared when you unplugged it
 - b. It should have a form similar to “COM8” (if working from a PC)

4. Once the board and serial port have been selected, we are ready to upload our program, or “sketch” as it is called in Arduino
5. Navigate to “File->Examples->OpenCM9.04->01_Basics->b_Blink_LED”
6. Open this sketch and verify your program looks similar to Figure 3.6



Figure 3.6-OpenCM Blink Code

7. Click on the right pointing arrow button in the top left of the Arduino IDE
 - a. When you hover over it, the text “Upload” should appear to the right
 - b. When you click this button, a status bar should appear in the bottom right of the IDE that will show the status of compiling and uploading
8. Look back at the OpenCM controller—the LED next to the word “STATUS” should be flashing on one second intervals

9. If this does not work, or things did not appear in the proper location, please return to sections 3.2 and 3.4 to confirm everything is set up properly. It is possible that several drivers may be missing or unavailable to your operating system.

Chapter 4: Dynamixel Programming

This chapter will introduce you to the low level concepts of programming Dynamixel motors, all the way down to sending individual bytes of data that make up packets of instructions. It will start at the lowest level and work up to higher level programming using the Arduino IDE with existing libraries to make control much simpler. Understanding the low level control will be useful when using higher level libraries, and it will be necessary to extend or create your own libraries for control.

More information about libraries is linked in section 2.1, but I want to take a quick aside to explain them in just a little bit more detail. Essentially a library in computer science is a collection of specific implementations of behavior that can be used over and over again in different contexts. As you read through this chapter, imagine how complicated it would be if you physically had to write out each byte of data you wanted to send to Dynamixel motors. Writing libraries allows you to write a complicated procedure once, leave it in terms of modifiable variables, and use it to send complex commands in just a few lines of code. Understanding the low level functionality is important to effectively create libraries for Dynamixel control.

4.1 Introduction to Low Level Dynamixel Programming

Dynamixel motors are controlled by receiving instructional packets, and sending status (or return) packets. A more detailed description of how this works can be found [here](#), but this section will attempt to highlight the relevant components for control with the OpenCM microcontroller.

This section is important for writing your own libraries later. For extremely basic control of Dynamixel motors, this section is not as relevant; but as soon as you want to do more complicated maneuvers with the motors this section is necessary.

4.1.1 Binary Data and Decimal Conversion

Dynamixel servos are commanded by receiving packets of binary instructions. You are controlling the motors by sending 0s and 1s to them, in a specific order; the motors will interpret the packets and perform what you command them. An analogy that I have found helpful when thinking of this communication is to think of each package of data as a sentence. One bit represents one 0 or 1, and this can be thought of as the letters that make up our sentences of data. One byte (8 bits) can be thought of as a full word. Each byte is made up of eight 0s and 1s in any combination that can be [converted](#) into integer data. This is important for understanding how to successfully utilize existing Dynamixel libraries. Perhaps libraries will be created in the future that will make it simpler to send data without having to worry about the 0s and 1s; but in the current state, the binary data is still somewhat relevant.

If one bit is a letter, and one byte is a word, then a package can be thought of as a sentence, or a command. It is comprised of several bytes of data in [machine language](#) that the Dynamixel will understand and interpret. Dynamixel motors will only understand a specific format of packets, so you need to follow the proper syntax.

The technical term for a “sentence” of data, or a collection of bytes, is a **packet** of data. When working with Dynamixel motors, we will send instructional packets, and the motors will then send status packets.

Another thing to consider when working with Dynamixel motors is that we will almost always be working with positive numbers – more specifically positive integers. The term for this in computer science is an unsigned integer. An unsigned integer is always a non-negative number. A computer can interpret a wider range of positive numbers with less data when it knows to expect unsigned values; this is because the first bit in a signed integer is typically reserved for the sign (positive or negative). An unsigned integer does not have to reserve this bit for the sign, so it can utilize all bits of data to represent the number. More on [signedness](#) can be found at the website linked.

The maximum value of a single byte is 1111 1111. As an unsigned integer, this is represented in decimal as 255. Refer back to the [conversion](#) instructions to learn how to convert an 8 bit binary number to a decimal integer. The minimum value of a single byte is 0000 0000, which in decimal is just 0. What this means is that each word (or byte) that makes up our packet of data is limited between 0 and 255.

4.1.2 Hexadecimal Representation

Another way to represent binary data is through [hexadecimal](#) conversion. This is a base 16 numerical system rather than base 2 (binary) or base 10 (decimal). The advantages of hexadecimal are that it is much simpler to convert binary to hexadecimal, and one byte of data can always be represented by two hexadecimal characters. When converting binary to hexadecimal you can break up the binary number into 4 bit values, and convert each 4 bit value into its hexadecimal representation. When you have converted each 4 bit value, you just concatenate (put together) the converted values to get the resulting hexadecimal number.

This is convenient because the maximum decimal number you can get from 4 bits is 15, and in binary this is represented by 1111. The range for hexadecimal is 0 to 15, but 10, 11, 12, 13, 14, and 15 are represented using A– F. So 1111 in binary (15 in decimal) maps to F in hexadecimal. The result is that each 4 bit number can be represented by a single hexadecimal character (0 to F). So a one byte value (8 bits) can be represented by 2 hexadecimal values. A 2 byte (16 bit) binary value can be represented by 4 hexadecimal values, and so on. When writing libraries to send packets of instruction to Dynamixel motors, it is often convenient to use hexadecimal values.

4.1.3 Dynamixel Communication Overview

Now that we understand how binary, decimal, and hexadecimal numerical representations can be converted, we can look at how we physically communicate with Dynamixel motors. We have already discovered that in order to control Dynamixel motors at the lowest level, the data must be sent in byte packets. Typically when building our packets of data, we will use hexadecimal representation. Again, the advantage of using hexadecimal to write out our data is that each byte takes up only two characters.

The flow of the data in Dynamixel motors is visualized in Diagram 4.1. The user’s code assembles an instruction packet using syntax the motors can interpret. This is explained in detail in 4.1.4. This packet is sent to the microcontroller (this text uses the OpenCM controller). The controller then broadcasts the packet to the motor(s) with the ID specified in the instruction packet. There is a generic broadcasting ID that will send the command to **all** connected motors. Once the motor(s) receives the packet, it executes the command that is specified in the packet. A status (or return) packet is then assembled and sent back

to the microcontroller. You can then use the program on the microcontroller, or a PC connected to the microcontroller, to interpret the status packet and complete necessary commands based on the status.

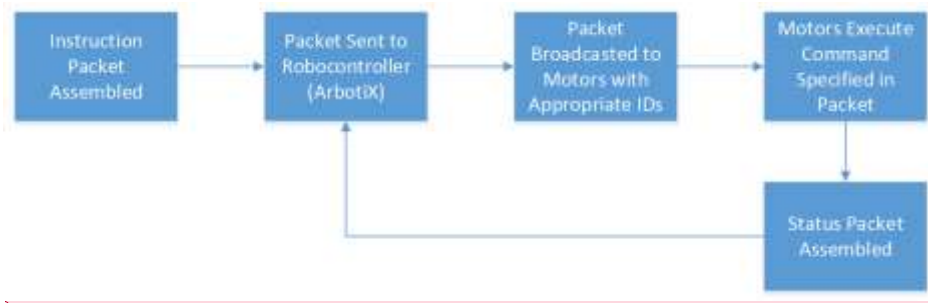


Diagram 4.1 - Dynamixel Data Flow

Keep the flow of data in mind while reading through the next two sections; the next two sections will cover instruction and status packets in much greater detail. You will learn the format in which these packets are sent, as well as what the hexadecimal values actually map to.

4.1.4 Dynamixel Instructional Packets

An instruction packet is command data that is sent to the main controller, which then sends it to Dynamixel motors. The structure of an instruction packet is shown in Figure 4.1. More detailed information regarding Dynamixel instructional packets can be found from [Robotis](#), but a summary can be found in this section.

0XFF 0XFF ID LENGTH INSTRUCTION PARAMETER1 ... PARAMETER N CHECK SUM

Figure 4.1-Format of Dynamixel Instructional Packet

The first two bytes that will make up our instructional packet of data will **always** be 0XFF 0XFF. I know in the previous section I said bytes represented in hexadecimal can always be represented by two characters, but that was only partially true. The physical value is represented by two characters (in this case FF); the 0X tells the microcontroller to expect a hexadecimal value following it. The FF in hexadecimal is equal to 255 in decimal, or 1111 1111 in binary. What the 0XFF 0XFF does is notifies the controller that it is the beginning of the packet. The binary equivalent of 0XFF 0XFF is 1111 1111 1111 1111. You can see that the binary representation is lengthy; this is why hexadecimal is the preferred numeric representation.

The next number in the packet is the ID. Dynamixel motors have a default ID value of 1 (out of the box), and can be set to anything between 0 and 253 (0X00 – 0XFD in hexadecimal). You can use a value of 254 (0XFE) to execute the command of instruction to **all** linked Dynamixel motors. This is known as the broadcasting ID. What this byte does is tells the controller which motor (or motors if using the broadcasting ID) to send the instructions to.

The next byte of data in the packet signifies the length of packet, which is going to be equal to the number of parameters + 2. The +2 comes from the form of the instruction, and from the checksum at

Commented [AH1]: Needs update such that 2nd block reads Packet sent to microcontroller (OpenCV)

the end of the packet. This lets the controller know how many additional bytes of data to expect as it executes the packet.

The following byte is reserved for the type of instruction to send. A table of the instruction types that can be sent can be found at the Robotis link above, or in Table 4.1 below. We will get more into the specifics of the instruction types in an example.

Table 4.1 - Instruction Data

Byte Value	Name	Function	No. of Parameters
0x01	PING	No execution. It is used when controller is ready to receive Status Packet	0
0x02	READ_DATA	This command reads data from Dynamixel	2
0x03	WRITE_DATA	This command writes data to Dynamixel	2 or more
0x04	REG WRITE	It is similar to WRITE_DATA, but it remains in the standby state without being executed until the ACTION command arrives.	2 or more
0x05	ACTION	This command initiates motions registered with REG WRITE	0
0x06	RESET	This command restores the state of Dynamixel to the factory default setting.	0
0x83	SYNC WRITE	This command is used to control several Dynamixels simultaneously at a time.	4 or more

The next N bytes of data to be sent are the parameters of the instruction. For example, if we were to write data to a motor, the parameters would have to include at least two things: first, we would have to send data that represents the property of the motor we want to modify (position, maximum torque output, ID number, etc...), and second, we would have to send the value that we want that property to contain. So if we wanted to change a motor's ID value, the first parameter would be the value that represents the ID property, and the second parameter would be our new ID value. Each Dynamixel motor has a similar core mapping of properties, but different series' may vary. You can find the tables that show each readable and writeable property for each series at the following [link](#) (navigate to the model of interest). For convenience, I have included the table for the AX-12A motors in Appendix 3. An explanation of how to read the tables is also provided in Appendix 3.

The last byte of data is the checksum. The checksum checks to see if the packet was corrupted upon transmission to the controller. It adds the values of all of the previous bytes of data and does some post-processing to avoid errors in most cases. The checksum **does not** include the first two bytes (0xFF 0xFF). Obviously, the checksum value could be (and will often be) greater than 255 (or 0xFF). If this is the case, then it is not possible to store this value in one byte, so the checksum does not **just** add the value of all previous bytes of data. It adds them all up, takes the lower byte of this added value (in hexadecimal it will always be the last two characters), and takes the opposite value. The way the opposite value is taken is by using the not bit operator ("~") which essentially takes the binary value, and every time it encounters a 0 it replaces it with a 1, and every time it encounters a 1 it replaces it with a 0. We will

examine this in an example in the next section. It is not 100% effective because of the nature of sending data in individual bytes, but it is effective most of the time.

The checksum is calculated both upon sending and receiving of the packet, and then the two values are compared. The value that is calculated prior to sending the packet is the last physical byte of the packet. Upon receiving the packet, it is calculated again and compared to that last byte value.

4.1.5 Dynamixel Status (Return) Packets

One of the really powerful things that Dynamixel motors do after executing the commands specified in the instruction packet is send a packet of their current status back to the controller. The format of the status packet is nearly identical to that of the instruction packet; the only difference is in place of the “Instruction” byte, the status packet sends back an “Error” byte. This format is shown in Figure 4.2. More detailed information can be found on the [Robotis](#) site.

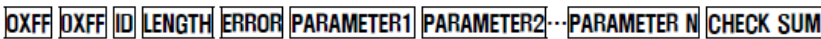


Figure 4.2 - Format of Dynamixel Status Packet

Again, the 0XFF 0XFF indicates the beginning of the packet, and the ID byte indicates which motor is transferring the status packet. The length byte is again the length of the packet, and can be calculated as $N + 2$ (the number of parameters) + 2. The +2 comes from the error byte and the checksum byte; the main difference between the instruction packet and the return packet is the instruction byte versus the error byte.

The next byte is the error byte. There are 7 potential causes of error in the Dynamixel motor, summarized below in Table 4.2, and also found on the Robotis support [site](#).

Table 4.1 – Example Status Packet Error Byte Definition

Bit	Name	Contents
Bit 7	0	-
Bit 6	Instruction Error	In case of sending an undefined instruction or delivering the action command without the <code>reg_write</code> command, it is set as 1.
Bit 5	Overload Error	When the current load cannot be controlled by the set Torque, it is set as 1.
Bit 4	Checksum Error	When the Checksum of the transmitted Instruction Packet is incorrect, it is set as 1.
Bit 3	Range Error	When a command is out of the range for use, it is set as 1.
Bit 2	Overheating Error	When internal temperature of Dynamixel is out of the range of operating temperature set in the Control table, it is set as 1.
Bit 1	Angle Limit Error	When Goal Position is written out of the range from CW Angle Limit to CCW Angle Limit, it is set as 1.
Bit 0	Input Voltage Error	When the applied voltage is out of the range of operating voltage set in the Control table, it is as 1.

The error byte is capable of identifying any of these potential causes of error using only one byte of data. For example if I had an error due to overheating, my error byte would look like 0010 0000. In this case, all bits are set to 0, except bit 2 (starting from bit 0) which signifies an overheating error. **Note:** This table represents an example of what each bit represents in the error byte, and this may be slightly different for different models. To determine exactly how to interpret the error byte for your model, reference the [manual](#) specific to the model you are using.

The parameter bytes represent the data that the motor is sending back to the controller. This would be used if you wanted to determine the current position of a motor. You could send an instruction packet command that would be to read the data of the position and this data would be sent back in the status packet using the parameter bytes.

The checksum byte works the same way in the status packet as it does in the instruction packet.

4.1.6 Dynamixel Packet Examples

Additional examples can be found [here](#), but we will go through two examples in this text before getting into higher level control. These examples are slightly simplified; in reality you are sending this data into the serial buffer of the robocontroller which will ultimately be sent directly to the motors, but the following demonstrates the format of the data that is being passed back and forth. Some more information about the buffer can be found [here](#).

Example 1: Write Data

How would we change the ID of an AX-12A motor from 1 to 5? Let's work through the packet of data we would have to send to write the new value of the ID (5) to the motor in the correct [EEPROM](#) address. Refer to Appendix 3 for the detailed difference between EEPROM and RAM.

Step 1: Beginning of packet

We know each packet will begin with the two bytes 0xFF 0xFF to signify that it is indeed the beginning of the packet. Our example is no different. So the beginning of our packet will just be 0xFF 0xFF.

Packet so far: **0xFF 0xFF**

Step 2: ID

The next byte of data that will make up our packet is the ID of the motor that we are writing data to. In this case, we are sending data to the motor with ID #1, so the value of this byte should just be 1. For consistency, we will represent all bytes of data in hexadecimal format. Our ID byte will be 0x01.

Packet so far: 0xFF 0xFF **0x01**

Step 3: Length

The next byte of data will tell our motor with ID #1 how many more bytes of data to expect. In this case, we will need two parameters (one to tell the motor that we are changing the ID, and one to tell the motor the value we are changing it to); we will also need one byte for the instruction, and one byte for the checksum. So our length byte will be 4 (in decimal) or 0x04 in hexadecimal.

Packet so far: 0xFF 0xFF 0x01 **0x04**

Step 4: Instruction

Our instruction byte is to tell the motors what we are doing – it will signify whether we are writing data, reading data, simply pinging the system to inform the controller of system status, etc... All possible instructions for Dynamixel motors are listed in Table 4.1. If you refer back to the table you will see WRITE_DATA is signified by the value 3 (0X03), and this is the value we will want to use to write a new ID.

Packet so far: 0XFF 0XFF 0X01 0X04 **0X03**

Step 5: Parameters

Now we need our two parameters. I mentioned in step 3 that the parameters would map to the address of the ID value in the EEPROM. In this case, we want to refer to Appendix 3 to see what that value is for the AX-12A motor. In this case it maps to address 3 (0X03) so our first parameter byte will be 0X03. Our next parameter is the value that we want to be stored in that address. Our goal is to change the ID of the motor to 5, so we want to write the value 5 to the address that represents ID, which we just discovered was 3. So our second parameter (5) will be 0X05.

Packet so far: 0XFF 0XFF 0X01 0X04 0X03 **0X03 0X05**

Step 6: Checksum

The last byte we have to send is the value of the checksum. This is probably the most difficult value to think about physically, but programming it is not as hard as it would seem. The checksum is equal to the opposite value of the low byte of the sum of all of the previous bytes. Let's go carefully through this example.

Step 6a: Add up all data bytes (not the 0XFF 0XFF):

$$0X01 + 0X04 + 0X03 + 0X03 + 0X05 = 1 + 4 + 3 + 3 + 5 = 16$$

Step 6b: Convert to binary

$$16 = 2^4 = 00010000.$$

Step 6c: Take the opposite value

In computer science there is a not bit operator that reverses the individual bits of a byte. In C++ it is designated as ~ (not). What this will do is change each 0 in the byte to a 1, and each 1 in the byte to a 0. So we will take our binary sum that we just found and apply the not bit operator to it.

$$\sim(00010000) = 1110\ 1111.$$

Step 6d: Convert to hexadecimal and compile the final byte

1110 in decimal is 14, in hexadecimal that maps to E. 1111 in decimal is 15, in hexadecimal that maps to F. So 1110 1111 = 0XEF.

Final Instruction Packet: 0XFF 0XFF 0X01 0X04 0X03 0X05 0X05 0XEF

The checksum byte can be recalculated by the controller and compared the value that is sent. If they are mismatched, that means some data was lost in communication and the data may be corrupted.

Example 2: Read Data

For our read data example, let's say we want to read the present voltage that the motor with ID 1 is receiving. Again, we will assume our motor is an AX-12A so we can use the table in Appendix 3.

Step 1: Beginning of packet

It is still just 0XFF 0XFF; all packets will start this way when working with Dynamixel motors.

Packet so far: **0XFF 0XFF**

Step 2: ID

The ID of the motor in which we are reading data is just 1, so our ID byte is 0X01

Packet so far: 0XFF 0XFF **0X01**

Step 3: Length

In this case we will have two parameters, plus the instruction and the checksum. Our length will be 0X04. The two parameters are the address of the voltage, and the length of the data that we are reading (1 byte). We will get more into this in step 5.

Packet so far: 0XFF 0XFF 0X01 **0X04**

Step 4: Instruction

Since we are reading data now, we can refer back to Table 4.1 and determine what value represents reading data. If you look back at the table you will see that read data is designated by 0X02, so our byte for this instruction is 0X02.

Packet so far: 0XFF 0XFF 0X01 0X04 **0X02**

Step 5: Parameters

The first parameter as I mentioned in step 3 is the address of data to be read. This will always be the start address, so if reading from multiple addresses (for instance if you wanted to read both bytes that signify position), the first parameter will always be the starting address. In this case, we are reading voltage being supplied to the motor, and our address (found in Appendix 3) is 42 (or 0X2A). The second parameter will be the number of bytes of data that we are reading. Since voltage is represented with only one byte of data, our value for this byte is just 1 (or 0X01). If we wanted to read from multiple addresses, we would need an additional byte for each address.

Packet so far: 0XFF 0XFF 0X01 0X04 0X02 **0X2A 0X01**

Step 6: Checksum

Now we are going to want to calculate our checksum value and attach it to the end of our packet so that the controller can verify that no data was lost upon sending the packet. We will take the same steps that

we did in example 1.

Step 6a: Add up all the data bytes (excluding the 0xFF 0xFF beginning of packet)

$$0X01 + 0X04 + 0X02 + 0X2A + 0X01 = 1 + 4 + 2 + 42 + 1 = 50$$

Step 6b: Convert to Binary

$$50 = 0011\ 0010$$

Step 6c: Take the opposite value

$$\sim(0011\ 0010) = 1100\ 1101$$

Step 6d: Convert to hexadecimal and compile the final byte

$$1100 = 12 \text{ in decimal which maps to C in hexadecimal}$$

$$1101 = 13 \text{ in decimal which maps to D in hexadecimal}$$

Checksum byte = 0XCD

Final Packet: 0xFF 0xFF 0X01 0X04 0X02 0X2A 0X01 0XCD

If these examples are still difficult to comprehend, more examples can be found at the site listed at the beginning of the examples, which is reproduced [here](#).

4.2 Programming Dynamixel Motors Using the Arduino IDE and OpenCM library

From section 4.1, it is pretty apparent that programming Dynamixel motors is not a simple matter, especially if we always had to compile individual packets. Fortunately for us, one extremely useful library already exists that makes programming the motors much simpler than compiling and sending physical bytes of data. The library does all of this for us behind the scenes, so we do not have to do it each time. There are still many limitations to the existing library, so understanding the low level data being sent to the controller is crucial to create your own libraries and extend existing libraries, but more of that will come in chapter 5.

The way this section will work is by doing a quick introduction to terminology and the Arduino environment, and then get right into practical examples of controlling the Dynamixel motors using the OpenCM controller.

4.2.1 Brief Glossary of Terms for Arduino

Some of these terms have been simplified for our specific uses, but more information can be found at the links.

[Sketch](#) – Another name for a program that is run on an Arduino microcontroller

[Library](#) – A collection of implementations of behavior that can be included in your sketch with the #include tag, which can be used throughout your sketch.

[Function](#) – A segment of code that allows a developer to modularize their sketch, which can be used in other functions by the appropriate function call.

Parameter – A parameter is a special kind of variable that is passed into a function that the function will reference to fully execute.

Register – Another name for the address of a specific property of a motor.

Serial Port – A connector by which a device sends one bit of data linearly at a time.

Baud Rate – The rate of data transfer between devices

4.2.2 Arduino Basics

Arduino was already introduced in section 2.1; if you do not remember the introduction I recommended rereading this short section or reading the following [introduction](#) provided by Arduino. This section will get into the basic formatting of an Arduino program, or “Sketch.”

Each “Sketch” for an Arduino controller will have at least two required functions: the `setup()` function and the `loop()` function. Additional functions can be created and utilized, but these two functions must exist for the sketch to compile and upload to the board. Any code that is placed within the `setup()` function will execute immediately upon uploading the sketch to the board. Any code that is placed within the `loop()` function will execute immediately upon completion of the `setup()` code; this code will continue to run over and over again until power is removed from the board, or another sketch is uploaded.

Arduino sketches are programmed using a set of C/C++ libraries, so if you are familiar with these languages, creating Arduino sketches should be really familiar to you. The [Arduino](#) webpage is well organized and should be referenced frequently with confusions regarding how they work.

This set will be using the library from the Dynamixel Workbench. Setting some common properties, such as the goal position and goal speed, have dedicated helper methods. All functions can be found on this API page: http://emanual.robotis.com/docs/en/software/dynamixel/dynamixel_workbench/. In this document, we will walk through setting the position of the motor and changing the ID.

Setting other properties can be set with the `dxl_wb.itemWrite` method. The list of properties that can be changed is listed in `dynamixel_item.cpp`, sorted by servo model number. This and other core libraries for the openCM9.04 are installed to
`...\\Arduino15\\packages\\OpenCM904\\hardware\\OpenCM904\\1.1.0\\cores\\arduino.`

4.2.3 Preparing the Hardware and Sketch

Before we can upload any programs or commands to control our Dynamixel, we will have to set up our hardware and prepare our sketch. This section will be important anytime you want to create a new sketch, and after doing it several times it will become nearly second nature.

Step 1: Ensure the Hardware is Connected Properly

Refer back to section 3.5 and make sure you can properly get the OpenCM controller to blink. Once the controller is blinking properly, you can connect the Dynamixel motor using the 3 pin Dynamixel compatible cable. In order to ensure the motor has power to run, it needs an external power supply – the AX/MX power hub. To hook everything up, use one 3 pin Dynamixel cable to connect the OpenCM microcontroller to the power hub (you may plug in to any of the 6 connections). Then, use another 3 pin cable to connect the motor to the power hub, again you may plug in to any of the 6 connections. These

ports are electronically equivalent; they can be used for organizing separate chains of motors but they will all behave the same. Finally, plug in the power hub using the 12V wall power adapter. This is similar to the figure below, but with the object in the top right being replaced by the OpenCM microcontroller.



Commented [AH2]: Replace picture with similar where computer adapter on right is the OpenCM

Step 2: Open Arduino and Setup the Sketch

Open the Arduino IDE application that was downloaded in Chapter 3. When Arduino opens by default it opens the last used sketch. Let us create a new sketch using "File->New". The name of the sketch will be something like "sketch_(today's date)". Let us resave the sketch to "Dyna_PositionSet" by going to "File" -> "Save As" -> "Dyna_PositionSet." We will use this sketch again in the next section to set the position of a Dynamixel motor. You can save it wherever you like, but I recommend creating a directory for all of your sketches. It may be worthwhile to dedicate a folder to "Learning" sketches as these will be useful to reference later on.

Now that our sketch is created and saved, we want to import the necessary libraries. Go to "Sketch" -> "Include Library" -> select "DynamixelWorkbench". If this library is not appearing, the steps in section 3.2 may not have been completed correctly; check back to 3.2 to make sure everything is in order. As soon as you import this library you should see line of code at the top of your sketch:

```
#include <DynamixelWorkbench.h>
```

Figure 4.3 - Arduino Library Import

This line of code is a header file (.h extension) inclusion that list the methods with parameters that are coded in a C++ file in the background. This is one of the key ingredients to a library that will be explored more in Chapter 5.

Step 3: Create and 'instance' of DynamixelWorkbench, dxl_wb

Now we need to create an object that refers to the dynamixel motors. This object will enable us to call specific commands that the motors are capable of executing, and give us a 'nickname' to reference the sending of these commands. Below the include line, type "DynamixelWorkbench dxl_wb;"

```
DynamixelWorkbench dxl_wb;
```

Step 4: setup() and loop()

Now we have to create our two necessary functions for our Arduino sketch. The first is the setup() function which will execute immediately after the sketch uploads; the second is the loop() function which will continue to execute on the Arduino controller until something interrupts it.

It is likely these functions were automatically created when you made a new file. However, if they were not, to create these two functions, we will go into our code below the library inclusions. First we will write void setup() followed by two curly brackets {}. We will then do the same for the loop() function immediately below it.

As you see, the words void, setup, and loop automatically have some formatting done to them. This is because they are Arduino recognized words that have a specific meaning for your sketch. This also means you cannot use these words as a variable name. They are reserved for their specific functionality.

Step 5: We now set up the connection between the OpenCM microcontroller and the dynamixels. In the setup section, we type the following: "dxl_wb.init("1",1000000);".

```
dxl_wb.init("1",1000000);
```

Here the string "1" is the serial channel that communicates with the dynamixels. Note: Most example sketches include an #ifdef structure to define DEVICE_NAME. This is the serial channel over which the control board interfaces with the Dynamixels. For the openCM9.04, this should be a string: "1". Some of the example sketches set it to "3", which results in the motors not communicating with the board, and it must be changed to "1". The 1000000 is the baudrate for communicating from the OpenCM microcontroller to the Dynamixels. The baudrate should be a standard value. The maximum for the AX servos is 1000000 (1 million). This value is programmed into the motor, and if it is not set correctly, the motors will not communicate. For changing this value on the motor, see [Appendix Not Made Yet](#).

Commented [AH3]: Make appendix for this.

Finally, after initiating the dynamixels, we need to ping them to enable us to send commands and read data from them. We do this with the following code:

```
uint16_t model_number = 0;
dxl_wb.ping(2, &model_number);
```

The result will look like the following in Figure 4.4:



Figure 4.4 - Arduino Sketch Setup

4.2.4 Changing the Position of a Dynamixel Motor

Now we are ready to control the motion of a Dynamixel motor. We are going to use our already setup sketch that we created in 4.2.3. With that sketch still open, complete the following steps. **Note: This section assumes you are working with a new AX-12A motor with an ID of 1. If this is not the case, refer to Appendix 2.2 to manually change the ID back to 1.**

Step 1: Center the Motor

Within the curly brackets of the setup() function, and after the dxl_wb.init, write the following lines of code:

```

dxl_wb.goalPosition(1, (int32_t)512);
delay(3000);

```

goalPosition is one of the defined functions in the DynamixelWorkbench library. All of the defined functions and variables can be found in the header file, either by opening it with the editor of choice on your computer or by viewing it [online](#). The goalPosition function will set the position of the motor with the ID represented by the first parameter to the value of the second parameter. The range of values for the position of an AX-12A motor are 0-1023, so setting it to 512 should center it. If you did not already know this, you may look to Appendix 3 and click on either of the "Present Position" links; this will give you the detailed information regarding the position address of the AX12-A.

The `(int32_t)` ensures that the function is using a number type that is a 32 bit integer. This is just like the discussion above on unsigned integers. Often functions require data to be of a specific type, and including this in our code here ensures that the function is receiving the number of the right type. This command 'casts' the following number into a 32 bit integer.

The `delay(x)` function will impose a pause on the execution of the code for x number of milliseconds, and it is essential for many Dynamixel operations (since the position cannot change instantaneously). This function is a default Arduino function.

Step 2: Move the Motor Back and Forth

In-between the curly brackets of the `loop()` function, write the following lines of code:

***NOTE: do not use the positions 0 and 1023 if a frame is attached to the motor. This will cause the motor to try to spin beyond its range of motion and will cause damage. With-frame limits can be found experimentally using Dynamanager. Be sure your frame is mounted correctly (points straight up while the servo is centered - note the line on the horn of AX-12As, and the notch in the shaft of MX-64Ts combined with the "pointing triangle dots" seen on their gold horns). For AX-12As with a properly mounted frame, values of 400 and 600 (instead of 0 and 1023) are well within the safe range of motion for this exercise.**

```
dxl_wb.goalPosition(dxl_id, (int32_t)0);
```

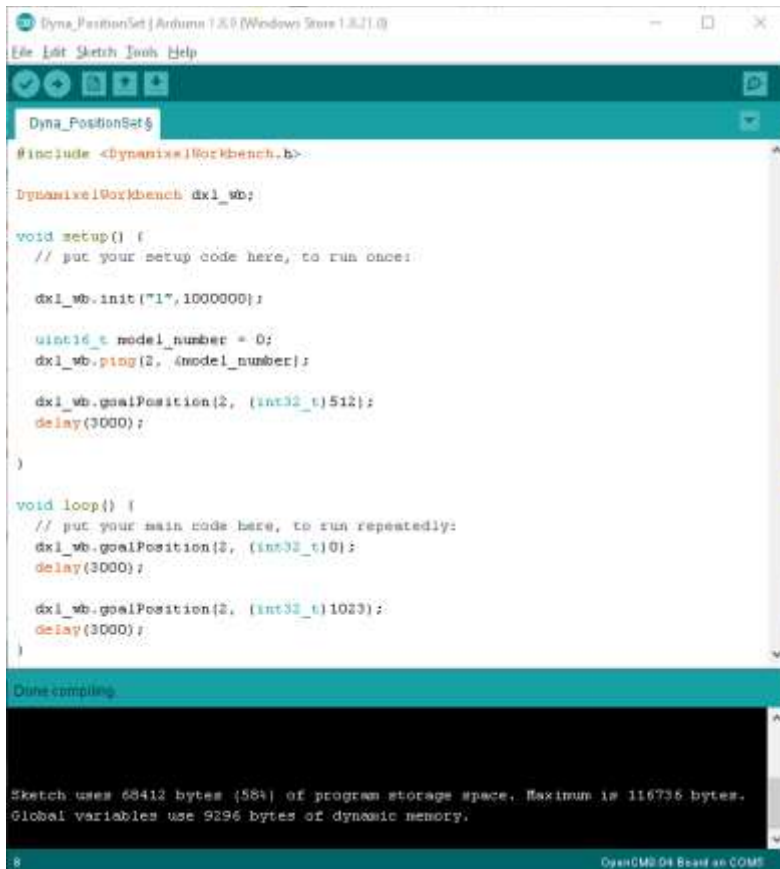
```
delay(3000);
```

```
dxl_wb.goalPosition(dxl_id, (int32_t)1023);
```

```
delay(3000);
```

These four lines of code will have the motor move to its two extreme positions with a three second delay in-between each motion.

Your code should look like Figure 4.5 below.



```

Dyna_PositionSet | Arduino 1.8.9 (Windows Store 1.8.21.0)
File Edit Sketch Tools Help

Dyna_PositionSet

#include <DynamixelWorkbench.h>

DynamixelWorkbench dxl_wb;

void setup() {
  // put your setup code here, to run once:

  dxl_wb.init("1", 1000000);

  uint16_t model_number = 0;
  dxl_wb.ping(2, &model_number);

  dxl_wb.goalPosition(2, (int32_t)512);
  delay(3000);
}

void loop() {
  // put your main code here, to run repeatedly:
  dxl_wb.goalPosition(2, (int32_t)0);
  delay(3000);

  dxl_wb.goalPosition(2, (int32_t)1023);
  delay(3000);
}

Done compiling

Sketch uses 68412 bytes (58%) of program storage space. Maximum is 116736 bytes.
Global variables use 9296 bytes of dynamic memory.




8 OpenCOM 04 Board as COM5

```

Figure 4.5 - Dyna_PositionSet Code

The line immediately before the “`dxl_wb.goalPosition(1, (int32_t)0);`” line is a comment. Comments can be written using the “`//`” designation, and any text written in a comment line will not be compiled and uploaded to the microcontroller. Comments are useful for explaining what the code will do, and helps maintain organization.

Step 3: Uploading the Sketch

To upload a sketch click the  (Upload) button in the top left of the IDE. If you just want to compile to check for errors, the  (Verify) button will do that for you; but the  will compile **and** upload. If there are any compiling errors, both will notify you, and the code will not upload until the errors are fixed. When you hit the upload button you should notice a status bar in the bottom right of the screen that will show when it is compiling, when it is uploading, and when it is done.

This program should cause the motor to center itself, and then continue to move back and forth with a three second delay between each directional change.

4.2.5 Getting and setting various properties

As we discovered in the last sketch, the `goalPosition(id, position)` function is useful for moving motors to a desired position. This next example will demonstrate two of the more versatile functions: `itemRead` and `itemWrite`. Both of these functions take in three parameters: the ID of the motor, the register (or address) of the property that you want to modify, and the value that you want to place in that register.

The `itemWrite` function will modify **any** value mapped to the table in the motor. This is useful for changing the ID of a motor, or modifying any value that is in the table. Names for table items can be found by going to the link [here](#), and clicking on the motor that you wish to control. Note that for properties with multiple words, use `_` in place of spaces (e.g. Goal Position becomes “Goal_Position” in the code).

itemWrite Example

Use the `itemWrite` function to change the ID of our motor from 1 to 2.

Step 1: Sketch Setup

First, create a new sketch and follow the example in 4.2.3 to setup the sketch. Make sure your `setup()` and `loop()` functions are identified properly. Also, make sure you import the `DynamixelWorkbench` library, initialize the system, and ping it so we can utilize the functions defined in the library.

Step 2: Set up variables

First, we will set a couple of variables for the current ID, and what you wish to change the ID to. We are doing this because we will need to use the values several times through code, and if you want to change and ID, it will be much easier to change the value once in the code, instead of hunting down all the correct places in the code. At the start of the setup code, before initiating the dynamixel communication, put in the following lines:

```
uint8_t currentID = 1;
uint8_t newID = 2;
```

And change the ping line to:

```
dxl_wb.ping(currentID, &model_number);
```

Step 2: `itemWrite(id, property, data)`

The name of this step is the definition of the `itemWrite` function exactly as it appears in the C++ code that defines it. The three parameters as mentioned above are the ID of the motor, the starting register value that you want to modify, and the data you are sending to that register (or address).

In order to change the ID of the motor from 1 to 2, we are going to need to know the property name that represents the ID for this motor type (referenced in the link above). We looked at this in section 4.1, but again we can refer to the link to find it. The property that represents the ID of an AX12 Dynamixel motor is “ID”. We now know two of the three parameters: the id is 1 and the property is “ID”.

Since we are changing our motor to have an ID of 2, our data will be 2. Changing the ID of the motor is something that only has to happen once for our code to be successful, so we will place our `ax12SetRegister` function, with the appropriate parameters, into the `setup()` function of the code.

The final code will look like this:

```

Dyna_IDChange | Arduino 1.8.9 (Windows Store 1.8.21.0)
File Edit Sketch Tools Help

Dyna_IDChange $

#include <DynamixelWorkbench.h>

DynamixelWorkbench dxl_wb;

void setup() {

    uint8_t currentID = 1;
    uint8_t newID = 2;

    //Initialize communication to the dynamixels
    dxl_wb.init("1",1000000);
    uint16_t model_number = 0;
    dxl_wb.ping(currentID, &model_number);

    //Change the ID from 1 to 2
    dxl_wb.itemWrite(currentID,"ID",newID);
}

void loop() {
    // put your main code here, to run repeatedly:
}

Sketch uses 69172 bytes (59%) of program storage space. Maximum is 116736 bytes.
Global variables use 9296 bytes of dynamic memory.
Enter bootloader
stm32ld ver 1.0.1
OpenCM Download Ver 1.0.4 2015.06.16
Board Name : CM-904

20 OpenCM9.04 Board on COM5
  
```

Figure 4.6 - Change ID of Dynamixel from 1 to 2

If we were to upload this sketch with our Dynamixel connected, the ID of the motor should change. We have several ways of testing this, but it is immediately unapparent if the ID actually changed or not.

Can you think of one way to test it?

Step 3: Testing to See if the ID Changed

One way to test to see if the ID changed is to upload the first sketch we created and see if the position moves. This is the most obvious way up to this point. If the ID changed successfully, our motor's position should not change. We would have to go into the code and change the ID of all of our `SetPosition(id, pos)` functions to have an ID of 2 instead of 1.

Another way to test it for immediate use in the sketch we just created is to use the `itemRead(id, property, &variable)` function. This function will get the value stored in the table and store it into the variable. So the three parameters are similar to that of `itemWrite`, with the exception of putting in where to read the data to, instead of what data to write.

In our `setup()` code, after the `itemWrite` call, ping the motor and then use the `itemRead` function to see if the ID actually changed. The line that will check the ID address is `itemRead(newID, "ID", &get_data)`. Notice, the `id` parameter changed to `newID`.

If we place that line of code into our sketch, it will not actually compile because we have not yet created the variable `get_data`. In order for `itemRead` to be of any use to us, we need to create a variable to store the value that we get from the function call. Depending on the size of the item in the table (number of registry spots), we will need different size variables to get the data. For ID, we will want to create an `int16_t` variable to store whatever it is that `itemRead` will return.

Let's add our variable to the beginning of our `setup()` function, where the other variables are, with the following line of code:

```
Int32_t get_data = 0;
```

In this line of code I created an `int` (integer) variable called `get_data` and I set it to equal 0. Since all values relating to a Dynamixel motor are greater than 0, we can initialize our variables to be 0 and it will never interfere with the values we set them to later. We can do if statements to see if nothing happened by checking if the variable is positive. If it is positive, we know it changed at some point, if not our code may be messed up.

Now our `get_data` should be storing the value 2 to represent the new ID of the motor. But how do we know that?

Step 4: Writing to the Serial Monitor

I'm sure most of you reading this who have done any computer programming in the past started with some sort of "Hello World" exercise where you displayed "Hello World" to the console. When working with Arduino, we do not have a console to write to. Instead, we write to our serial port, and we can use the "Serial Monitor" to see what is being written. In order for us to utilize it, we need to initialize our `Serial` object by using its constructor.


To do this in Arduino we simply write `Serial.begin(9600);` (where 9600 is the baud rate). This will usually be one of the first commands in our code, so we will place it before our `"dxl_wb.init("1",1000000);"` line.

Once our `Serial` object has been initialized, we can now do a lot with it. There will be more examples of the use of `Serial` in chapter 6, but for now we just want to write to the serial monitor. One other useful

convention is to put a delay in-between any Serial.write lines, as the data can only transfer so quickly. Let's write "Hello World" to our Serial monitor, and then we will write the newID value.

Step4a: Write "Hello World"

For those of you have coded in Java or C#, these next few steps will look somewhat familiar. To write a line to the Serial monitor without a carriage return, simply write `Serial.print(data)`. To write a line with a carriage return at the end, use `Serial.println(data)`, where `ln` means line. So, let's do just that; after the `Serial.begin(9600);` line, write `while(!Serial);` followed by `Serial.println("Hello World!");`. The `while(!Serial);` gives you, the user, time to open the Serial Monitor in Arduino once the code has uploaded, and won't continue with the rest of the code until you do. Once the upload completes, click the Serial Monitor button until it opens a new window.

Following the `Serial.println("Hello World!");` put another 10 millisecond delay. Now, upload the sketch, and when it is done uploading click on the  icon in the top right of the Arduino IDE. This will open the serial monitor. You should see "Hello World!" in the Serial Monitor.

Step 4b: Writing the Data to the Serial Monitor

Now we know how to write lines of text to the Serial Monitor, but what about data? It is just as easy as writing lines of text. Underneath the `dxl_wb.itemRead(dxl_id, "ID", &get_data);` line of code, write the following lines:

```
Serial.print("The new ID of the Dynamixel is: ");
Serial.println(get_data);
```

Upload the sketch to the board and open the Serial Monitor.

***Note:** Immediately setting, then reading a newly set value (as is done with back to back "itemWrite" then "itemRead" commands above) often does not work as expected. Though the ID value **IS** changed by uploading the sketch, it does not change instantaneously and the "itemRead" command may fail (since it doesn't yet see a motor of ID 2 when it is executed. You will know this has happened if the serial monitor prints 0. Adding a 50 ms delay between "itemWrite" and "itemRead" commands solves the problem.

Code Summary:

The total code for this sketch should compare to the following.

```

Dyna_IDChange
File Edit Sketch Tools Help

Dyna_IDChange$

#include <DynamixelWorkbench.h>

DynamixelWorkbench dxl_wb;

void setup() {

  uint8_t currentID = 1;
  uint8_t newID = 2;
  int32_t get_data = 0;

  //Initializes the serial monitor
  Serial.begin(9600);
  while(!Serial); // Wait for Opening Serial Monitor
  Serial.println("Hello World!");
  delay(10);

  //Initialize communication to the dynamixel
  dxl_wb.init("1", 1000000);
  uint16_t model_number = 0;
  dxl_wb.ping(currentID, &model_number);

  //Change the ID from old to new
  dxl_wb.itemWrite(currentID, "ID", newID);
  dxl_wb.ping(newID, &model_number);

  //Checks the ID property of the dynamixel
  dxl_wb.itemRead(newID, "ID", &get_data);

  //Prints the data to the serial monitor
  Serial.print("The new ID of the Dynamixel is: ");
  Serial.println(get_data);

}

void loop() {
  // put your main code here, to run repeatedly:
}

Ready To download
Flash : 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Write Size : 65812
Checksum : Success..
Go Application

OpenCM904 Board on COM5

```

Figure 4.7 - Change the ID from 1 to 2

A quick (and useful) exercise that I recommend doing now: modify the code in the last sketch to change the ID back to 1. Unless you are going to start chaining motors together, it is usually a good idea to keep the IDs set to 1 so you don't forget which is which.

4.3 Controlling Several Motors Simultaneously

Controlling multiple motors simultaneously is as simple as [daisy chaining](#) multiple motors together and writing commands to different IDs. It is important to make sure each motor in the chain has a unique ID, which can be set using the example in 4.2.5, or can be manually set using the instruction in Appendix 2.1. To daisy chain the motors you simply take a 3 pin Dynamixel compatible cable and connect it from the available port of one Dynamixel motor to another available port of a separate Dynamixel motor. The connection will be from the OpenCM controller or power supply port to one Dynamixel to the next Dynamixel etc... It is important to make sure your power supply is capable of providing enough current to each motor to get ample torque. It may be necessary to purchase a higher current rated power supply than the one suggested in this manual if you are chaining many motors. I also recommend physically marking each motor with the ID of the motor so you do not forget which ID is which. It is also good practice to chain motor with ID #1 closest to the controller, and then increase the ID by one for each one chained off of it. This is not required, but it makes it easier to remember.

Once you have multiple motors chained together, programming them is as simple as changing the ID values in your function calls. All you do is change the ID parameter for the function that you want to command the specific motor. So if I want to move motor with ID #3 to position 512, the command would simply be `dxl_wb.goalPosition(3, (int32_t)512);`. Recall, you can use the broadcasting ID to send a command to every motor in the chain. The broadcasting ID is 254, so if I sent the command `dxl_wb.goalPosition(254, (int32_t)512);` to the controller, every connected Dynamixel would try to move to position 512. You can use this knowledge when creating libraries to make sure you are sending commands to the appropriate motors. Most functions created to control Dynamixel motors will take in an ID parameter.

An example of controlling several motors can be found in section 4.4.

4.4 Writing Additional Functions for Modular Programming

When writing more advanced and complicated sketches, it is good practice to break off specific functionality into [functions](#). Functions are most appropriate for functionality that will be repeated in your own code, but may be unique to what you are trying to accomplish within the sketch. This is different from a library, which is useful code for a wide variety of applications. An example that will be used throughout this section is this: let's say we have a relatively simple robot with six Dynamixel motors, we want it to move around to different positions, and then we want it to return to its known "rest" position. In order to have the robot return to rest position, we have to set the position of each individual motor to its measured rest position value. The rest position of our robot is unique specifically to our design.

Now, let's imagine that we have our robot doing many different movements. If we had to manually set the position of each motor every time we want it to return to rest position, the code will get sloppy and filled with `SetPosition(id, pos);` lines. Every time we return it to rest we have six lines of code. One way to make this much cleaner is to create a separate function that sets the position of the entire robot to its rest position. Then, in the body of our code, every time we want it to return to rest we would only need one line (as opposed to six).

4.4.1 Simple Function Example

Let's take a look at how we would do this in the code. Create a new sketch called `FunctionTest`. Create your `setup()` and `loop()` functions, and import the `DynamixelWorkbench` library. Now, outside of both the `setup` and `loop` functions, create a void `setRestPosition()` function. The syntax will be identical to the `setup` and `loop` functions; the only difference is the actual function name.

The "void" at the beginning of the function is called the return type. Functions can return any data type (integers, strings, etc...), and you would designate what you want the function's return type with the first word of the function name. In this case, since we want our function to merely set the position of the six motors, we do not need a return type. When you create the `setRestPosition()` function, do not forget to include the open and close curly brackets ("{" and "}").

In-between the curly brackets, write the following lines of code:

```
dxl_wb.goalPosition(1, (int32_t)100);
dxl_wb.goalPosition(2, (int32_t)200);
dxl_wb.goalPosition(3, (int32_t)300);
dxl_wb.goalPosition(4, (int32_t)400);
dxl_wb.goalPosition(5, (int32_t)500);
dxl_wb.goalPosition(6, (int32_t)600);
delay(3000)
```

Our fictitious robot has a rest position when motor 1 has position 100, motor 2 has position 200, and so on. The delay at the end of the function gives the motor three seconds to return to this position. This is also the appropriate way to control more than one `Dynamixel` motor from one sketch. All we had to do was change our ID parameter of the `goalPosition` function.

Now, anytime you write the line of code "`setRestPosition();`" anywhere else in your code, these six lines will execute. You can use this code in the `setup` function, `loop` function, or even other created functions.

Your code should look like the following so far:



Figure 4.9 - Creating Functions in Arduino

Notice the comment above the name of the function describing what it does. This is good coding practice, for functions that are much longer it is really helpful to have a short summary above the function to explain to the programmer what it does. This is especially important if you are working on a project with multiple people.

If I wanted to setRestPosition() in my setup loop, the setup function would look like the following:

```
#include <DynamixelWorkbench.h>

void setup() {
  // put your setup code here, to run once:

  //Sets the robot to its rest position
  setRestPosition();
}
```

Figure 4.10 - Function Call

This line of code is known as the “function call” (or the call to the function). The parenthesis tell the compiler that this is referring to a later function, and to run that function in its entirety when it reaches the call.

4.4.2 Function with Parameters and Return Value

Now, let’s say we want a function that will measure the position of motor A, and adjust the position of motor B depending on the position of motor A. Then, we want it to return the value of the new position of motor B back to the code that called it. Motors A and B may be any two motors in our chain, depending on what we are trying to accomplish. Since I said I want it to return the value of the new position of motor B, I know I will need a return value. Since the position of a motor is always represented by an integer, my return type will be “int.” I also will need something to specify which of our motors is motor A, and which is motor B. For this, we will use parameters, or values that we include with our function call to tell our function which is which.

Let’s call the function “adjustPosition.” The function definition (or name of the function) is shown below:

```
//Set position of motor b based on position of motor a
//Returns the new value of the position of motor b
int adjustPosition(int a, int b)
{
}
}
```

Figure 4.11 - More Advanced Function Definition

The “int” before the name of the function lets us know that the function will send an integer back to wherever it is called. The “int a, int b” in-between the parenthesis means that there are two integer parameters that the function expects when we are calling it.

It is also possible to [overload](#) the function, meaning we can have multiple function definitions with the same name but different parameter signatures. In other words, I can create two functions with identical names, one with some parameters and one without, and the compiler is smart enough to distinguish between the two. You can do this as many times as you want as long as they do not have the same number and types of parameters.

Since a and b are parameters of our function, we can use them however we would like within the function. The first thing we will want to do is find the position of motor a and store it in a different integer. Assuming we are working with an AX-12A motor, that line of code would look like this:

```
//Gets the current position of motor A
dxl_wb.itemRead(a, "Present_Position", &aPos);
```

Notice the itemRead is itself a function, so we have been using function calls all along. The difference is that this function is defined in the library, whereas now we are defining the function in our current sketch. This line would get both bytes of the present position of motor with id "a" and store it in "aPos". Let's say if this position is less than 512, we want to set motor b to this same position. If it is greater than or equal to 512, let's set the position of motor b to 1023. We then want to return this value. As soon as you encounter a line of code with the word "return" in a function, the function will end. So, we can put the word return in a function in multiple places (depending on the logic ladder) and it will not break the function. In this case, I will put a return value in both the if statement, and the else statement. The final function code is as follows:

```
//Set position of motor b based on position of motor a
//Returns the new value of the position of motor b
int adjustPosition(int a, int b)
{
    //Gets the current position of motor A
    dxl_wb.itemRead(a, "Present_Position", &aPos);

    if(aPos < 512)
    {
        dxl_wb.goalPosition(b, (int32_t) aPos);
        return aPos
    }

    else
    {
        dxl_wb.goalPosition(b, (int32_t) 1023);
        return 1023
    }
}
```

Figure 4.12 - Parameter and Return Example

As you can see, if the position of motor a is less than 512, it will set the position of motor b to whatever it currently is. It then returns the value that b will get to. If it is greater than or equal to 512, it sets b to 1023 and returns 1023. Only one of these logic paths can be utilized in a function call, so only one integer will ever be returned.

Now, let's put it all together. In our setup() function, after we put our robot in its rest position, set the position of motor 2 based on motor 3. We want to store the new value of the position of motor 2 in the integer variable "pos2". See if you can put the appropriate function call in your loop, and compare to the final sketch below. Note that the posted code will not work on its own. Just like in previous examples, you will need to initialize any variables (aPos), add in the motor initialization code, and ping the motors before we can communicate with them.



```

FunctionTest | Arduino 1.8.9 (Windows Store 1.8.21.5)
File Edit Sketch Tools Help

FunctionTest

#include <DynamixelWorkbench.h>

void setup() {
  //Sets the robot to its rest position
  setRestPosition();
  //Adjusts position of motor 2 based on the position of motor 3
  int pos2 = adjustPosition(3,2);
}

void loop() {
  // put your main code here, to run repeatedly:
}

//This function will return the robot to its rest position
void setRestPosition()
{
  dxl_wb.goalPosition(1, (int32_t)100);
  dxl_wb.goalPosition(2, (int32_t)200);
  dxl_wb.goalPosition(3, (int32_t)300);
  dxl_wb.goalPosition(4, (int32_t)400);
  dxl_wb.goalPosition(5, (int32_t)500);
  dxl_wb.goalPosition(6, (int32_t)600);
  delay(3000);
}

//Set position of motor b based on position of motor a
//Returns the new value of the position of motor b
int adjustPosition(int a, int b)
{
  //Gets the current position of motor A
  dxl_wb.itemRead(a, "Present_Position", &aPos);

  if(aPos < 512)
  {
    dxl_wb.goalPosition(b, (int32_t)aPos);
    return aPos;
  }
  else
  {
    dxl_wb.goalPosition(b, (int32_t)1023);
    return 1023;
  }
}

Once Saved.

OpenCM304 Board. OpenCM304 Bootloader on COM5

```

Figure 4.13 - Function Test Sketch

END OF UPDATING FROM ARBOTIX-M to OPENCM

If you make it this far, some exercise to try using the tools above.

1. Move the motor around using itemWrite commands instead of goalPosition
2. Change the ID of a motor to 2, connect two motors to the hub (or daisy chain the motors), and send position commands to each motor.
3. Control an MX-64.
4. Chain in an MX-64 with two AX-12s and send commands to all 3 motors.
5. Adjust other properties. E.g. Change the velocity and try moving the motors at different rates.

For the extra-motivated, you can try the following by piecing together what you know and some of the document that follows. Some future updates to the document within the next couple weeks will spell out more clearly how to do these things.

6. Send commands from Matlab to the OpenCM, and move a motor based on that.
7. Move motors simultaneously.

Chapter 5: Creating and Extending Libraries

Libraries are extremely important. Without the existing DynamixelWorkbench library, we would have to go back to section 4.1 to send individual bytes of data to control the motors. That would not be easy. The existing DynamixelWorkbench library is really useful, but this chapter will show you how to create your own libraries.

This chapter assumes more experience with object oriented programming, and also that you have some IDE installed for C++ development. I use [Visual Studio 2010](#) Express, but any C++ IDE should be fine. If you choose to use VS 2019, you will need to create a free Microsoft user account with which to register the product. Registration is free, but must be done within 30 days of installation.

This chapter will try and teach you to create your own libraries by going through an example. The example we will use is to create a library that can control torque output based on an integer input that lies within the range 0 and 1023. We will use the ax12 library in the torque control library. **Note: This example will be a really simple example of how libraries can be created. Real libraries will be much longer and more powerful.**

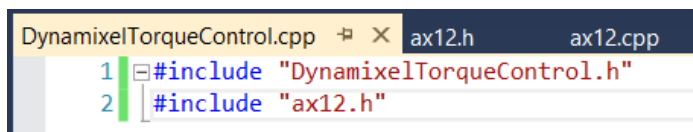
5.1 Create Your C++ File

In your IDE of choice, create a new C++ file. In Visual Studio, you go to “File” -> “New” -> “File” -> “Visual C++” -> “C++ File (.cpp).” Name your file DynamixelTorqueControl.cpp. Create a new folder in the “Arduino” -> “Libraries” folder (where you installed the Arduino IDE) called DynamixelTorqueControl and save this new C++ file in that folder. This will make it easy to import into our Arduino sketches as soon as we are done. The first lines of your C++ file will be the #includes (similar to how we did it in the Arduino IDE).

5.1.1 #include

When creating a C++ library, you must **always** include the corresponding header file. In this case, our corresponding header file will be “DynamixelTorqueControl.h” (it will always have the same name as the cpp file, but with a .h extension). We also want to use the ax12 library, so include that as well.

The code should look like this so far:



```

DynamixelTorqueControl.cpp  ax12.h  ax12.cpp
1  #include "DynamixelTorqueControl.h"
2  #include "ax12.h"

```

Figure 5.1 - #include Tags

Now, we want to start defining our functions.

5.1.2 Creating Functions

When creating a library for computing torque output, what functions do we want to have? The most obvious is a function to turn torque on in one direction at some magnitude. Dynamixel motors have two “modes” that can be switched by modifying the clockwise and counterclockwise angle limits. For now,

we are going to assume joint mode (which is the default). The other mode is wheel mode, and more can be found about that in the [Robotis E-Manual](#) for the ax12.

Before writing our torque output function, we need to consider the return types and the parameters. When we command our motor to apply a feedforward torque, do we want any values to be returned? We just want our motor to apply the torque, so our return type is void. Now we need to think about parameters. The function will need three parameters: the ID, the direction, and the value. Let's call those parameters id, direction, and magnitude. Each of these three parameters can be represented by integers, so they should be type "int." We will name the function "applyTorque".

In C++, our function declaration will look like this:

```
4 void applyTorque(int id, int direction, int magnitude)
5 {
6
7 }
```

Figure 5.2 - Function Definition

Now we want to make our motor apply the torque based off of those three parameters. Fortunately, since we include the ax12.h file, we can use the functions defined in the ax12 library to control our motor.

For AX12-A motors, the easiest way to estimate output torque is to set the maximum torque value and then set an extreme, distant position. When setting the position, the motor will apply as much torque as its limit will allow to get to that position.

5.1.3 Limiting Torque

In the Dynamixel memory, there are two separate locations to specify torque limits. One of these lies in the EEPROM and the other in the RAM. They effectively do the same thing, with the exception that the EEPROM value will save even on a power reset. The value in the RAM will be set relative to the value in the EEPROM. We will modify the value in the RAM so resetting it is easy (just requires a motor restart).

If you look in Appendix 3, you can find this value in addresses 34 and 35. Since it occupies two addresses, we will have to use the ax12SetRegister2 function. Since we are creating a library, we will want to use defined variables rather than hard-coded integers for our address. So, using our knowledge from chapter 4 lets write out our torque limiting line. Try this on your own, then look below to check your code.

```
void applyTorque(int id, int direction, int magnitude)
{
    ax12SetRegister2(id, AX_TORQUE_LIMIT_L, magnitude);
}
```

Figure 5.3 - Torque Limiting Code

Notice how every parameter is represented by a variable; there are no hardcoded values in this function call.

5.1.4 Applying Torque

So now that our torque limit has been set to the desired output value, we can simply just set the position to whatever we want and it will be torque limited to our magnitude. But a direction is a parameter, not position.

In this case, we can just use the two most extreme position values (0 and 1023) to set our position. That will effectively give it a direction to move until it can move no more. We can do this two ways: we can use an if-else ladder with the specified direction, or we can do it one line of code by using some math with the direction. If we specify in documentation that the direction will be 0 or 1, 0 for clockwise and 1 for counterclockwise, we can do our torque application in one line of code.

Your final applyTorque function should look like the following:

```
4 void applyTorque(int id, int direction, int magnitude)
5 {
6     ax12SetRegister2(id, AX_TORQUE_LIMIT_L, magnitude);
7     SetPosition(id, direction*1023);
8 }
```

Figure 5.4 - Full applyTorque Code

By extending the ax12 library into this library, we can achieve a lot of functionality in very few lines of code.

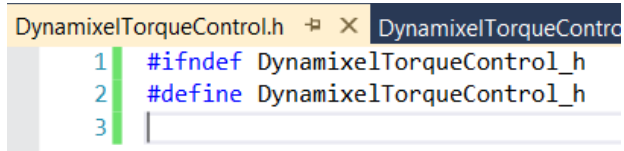
You can continue to add functions to this library similar to how we just made applyTorque, but for now let's take a look at the header file that will be tied to this C++ file.

5.2 Create Your Header File

Again, using your IDE of choice create a new header file. In Visual Studio go to "File" -> "New" -> "File" -> "Visual C++" -> "Header File (.h)" and hit "Open." Save it in the same folder you saved the C++ file, and with the same name (with a different extension). In this case it will be DynamixelTorqueControl.h.

5.2.1 #define and #include

Much like the C++ file, we will need to put a couple of definitions at the beginning of our header file. The first two will look the same for every header file, and I'll explain what it does after I show you what it looks like. The first two lines should look like this:



```
DynamixelTorqueControl.h
1 #ifndef DynamixelTorqueControl_h
2 #define DynamixelTorqueControl_h
3
```

Figure 5.5 - Header Definition

These two lines of code will first check to see if the library has already been defined, and if it has not been defined it will define it. This is important to include for the following scenario: say a different

library includes this library (much like our library includes the ax12 library), if both libraries are referenced in a sketch, you may get errors for trying to define the library twice. This prevents that from happening

Next, we will want to include the ax12.h library, just like we did in the C++ file. The line will be identical, it is just `#include "ax12.h"`.

5.2.2 Referencing Functions

The header file tells the code which functions you can actually call in your library, including return types and parameter types. In this case, our library only has one function, so that is the only one we will need to include in our header file. This will look exactly like the function definition in the C++ code, with the addition of a semicolon at the end. In this case, it will look like the following:

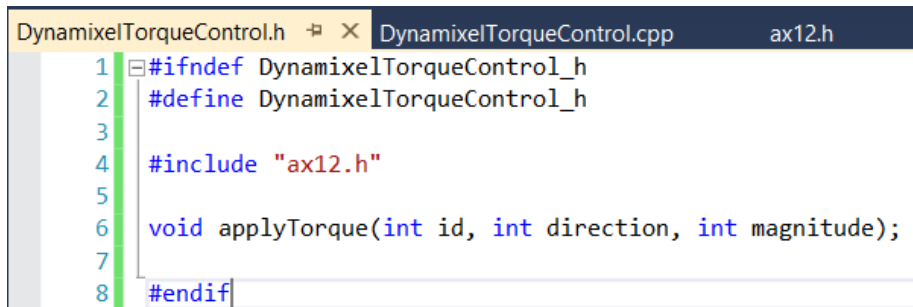
```
6 void applyTorque(int id, int direction, int magnitude);
7
```

Figure 5.6 - Header Function Reference

That is all our header needs to contain. You would repeat this step for any additional functions in the library, and you can also define variables in the library using `#define` (variable name) (variable value). You can look back to the ax12.h file to see many examples of this.

5.2.3 Ending the Header File

The last thing we need to include in the header file is a `#endif` statement. When we included the `#ifndef` tag, we essentially said "if this library is undefined, do the following:"; now we need to tell the code that it is done. To do that we just put a `#endif` statement at the very end of the header file. The entire header file should look like the following:



```
DynamixelTorqueControl.h  X DynamixelTorqueControl.cpp  ax12.h
1 #ifndef DynamixelTorqueControl_h
2 #define DynamixelTorqueControl_h
3
4 #include "ax12.h"
5
6 void applyTorque(int id, int direction, int magnitude);
7
8 #endif
```

Figure 5.7 - Full Header Definition

5.3 Using Our Library

Now that our library has been created and saved in a specific folder under "Arduino" -> "Libraries" we can now use it to program our motors. Create a new sketch in the Arduino IDE. Call it "DynaTorqueTesting." In the IDE go to "Sketch" -> "Import Library" -> "DynamixelTorqueControl." **Note:** If you have not closed and reopened Arduino since creating the "DynamixelTorqueControl" folder, this library will not appear. Simply close Arduino and open it again.

Also import the ax12.h library as well. You can do this by manually typing “#include <ax12.h>” or by importing the Bioid library as we have done in the past. If the ax12 library is not included as well, you will encounter compilation errors (since our library references the ax12 library).

Now that our library is imported into our sketch, we can try giving our motor a torque. Try it out for yourself! The syntax should look something like this:

```
applyTorque(1, 0, 500);
```

Try having it go back and forth in the loop function with appropriate delays and magnitudes, and see what you get! Just remember, the limits on the torque limit address are 0 and 1023, so make sure your magnitude parameter lies within that range.

Some other excellent resources to refer to when creating libraries is the [Arduino Library](#) page or simply looking at the ax12.h and ax12.cpp files that are in the libraries folders in your Arduino directory. It is easiest to learn from examples and modifying examples to suit your needs.

Chapter 6: Real-time Control Using MATLAB

So far we have learned how to control the Dynamixel AX12-A (the concepts transfer to other motors as well) motors by uploading sketches to an OpenCM controller, but everything we have done so far has been from predefined functions. Most applications though would prefer some sort of real time control. For example, if we wanted to create a remote controlled robot, what we have learned so far would not be sufficient.

The major concepts introduced in this chapter will be applicable to broader range of programming languages and applications, but we will use MATLAB for our examples. MATLAB has a well-defined set of built in functions that makes real-time control simpler than other languages.

This chapter assumes that you have access to MATLAB and have some familiarity programming in MATLAB.

6.1 Serial Communication Revisited

So far every time we have uploaded a sketch from our Arduino IDE to our OpenCM controller, we have sent the data over a serial connection. This is why there is a slight delay between “Uploading” the sketch, and having the program actually executed by the motor: the data can only transfer to the motor so quickly. This data transfer is limited by the baud rate. One limiting factor of serial communication is that it can only send one bit of data at a time, in a sequential fashion. So it is really only sending one 0 or one 1 at a time. Many implementations of serial objects have been designed to compile multiple bits to be sent in order, but there are still limitations to how much data can be sent at a time.

So far we have only used our serial connection to send fully compiled programs to our ArbotiX controller. But we can use it for much more. As you discovered earlier in this text, you can write to the Serial Monitor in Arduino by simple Serial.print and Serial.println commands; those are only two of the functions contained by the serial object. A full list of functions that the serial object is capable of executing can be found [here](#).

Unfortunately, the Arduino IDE is not currently capable of sending data over serial connection in real-time, so we will have to use another editor or program. This is where MATLAB excels (at least for PCs).

For Unix machines, sending data over serial connections using Python is relatively simple, and more information about that can be found [here](#), but for now we will focus on using MATLAB on a PC.

6.2 Sending Data through Serial Connection in MATLAB

Fortunately for us, MATLAB has a dedicated [serial](#) object (much like Arduino), but MATLAB can open the port to send data through the USB to FTDI cable (the serial connection). Using MATLAB to send data to the ArbotiX controller is therefore a pretty trivial task; however, we must also program our ArbotiX controller to interpret the data that it is receiving and do something with it. The flow of data is summarized in Diagram 6.1 below.



Diagram 6.1 - MATLAB-Arduino Serial Data Flow

The way we will discover these topics is a little out of order from how the data flows; we will start with uploading an Arduino sketch to our controller that is capable of interpreting data received from the serial port, but before we do that we are going to define our problem in great detail.

6.2.1 Defining the Problem (or Goal)

Since there are several components that go into creating a functioning and well-behaving serial controlled program, it is crucial to define the problem in great detail, including limitations. This section will do just that.

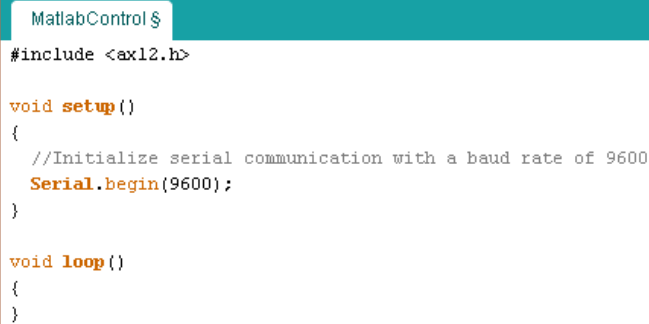
For this chapter, our goal will be to create a function in MATLAB that can send a positional value (ranging from 0 to 1023) to the ArbotiX controller which will be received and interpreted, and then sent to the motor with ID #1 to move to that specified position. We will then have our motor send its position back to MATLAB so we can verify our communication. Our biggest limiting component is that MATLAB can compile and send 8 data bits sequentially over serial at a time. So we can only send numbers between 0 and 255.

The solution to this limiting factor is to send two numbers that will make up the 2 byte number of 1023. We will see this implementation when we get to writing our MATLAB function.

6.2.2 Uploading a Sketch to Receive and Interpret Data

In order to receive data over serial communication, we are going to have to utilize many more of the implement serial functions (more than just `Serial.begin`, `Serial.print`, and `Serial.println`). Let's create a new sketch and call it `MatlabControl`. Save it with the rest of your sketches so far. Make sure to import the `ax12` library (either by importing `Bioid` or just using `#include <ax12.h>`). Also, create your `setup()` and `loop()` functions.

In your `setup` function, let's initialize our serial communication by the `Serial.begin` command. We will give it a baud rate of 9600 again, so the code will look like this:



```

MatlabControl$
#include <ax12.h>

void setup()
{
  //Initialize serial communication with a baud rate of 9600
  Serial.begin(9600);
}

void loop()
{
}

```

Figure 6.1 - Arduino Serial Communication Setup

That is all we will need for the setup function.

We want our loop function to keep checking to see if any new data has arrived. But where is that data going once it is sent to the controller? The data is stored on the board in what is called the [data buffer](#). Arduino has a built in function to see how many bytes of data are being stored in the buffer. For those in computer science, the buffer behaves like a [queue](#) data structure. A queue means that data is read from the buffer in the same order that is written to the buffer. The serial buffer in Arduino can hold up to 64 bytes of data, but for our goal (to write position to a motor), we will only need 2 (because the biggest number we will ever have to write is 1023, and this number can be represented in two bytes of data).

The built in Arduino function to check how many bytes of data are being stored in the buffer is a function under the Serial object, called `available()`. It returns an integer value that represents the number of bytes available to read from the buffer. Since we are looking for a two byte value, we will want to read data when `Serial.available()` is equal to 2. There are several ways to write this in the code, but the most intuitive is to have our program just loop and search for if there is more than one byte available in the buffer. In the code, it would look something like this:

```

if(Serial.available() > 1)
{
  |
}

```

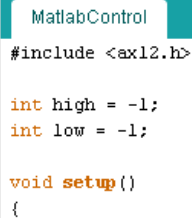
Figure 6.2 - Checking Buffer for Amount of Data

Now, whatever code we place within the curly brackets of the if statement will only run when there are two or more bytes of data in the buffer. The data will also stay in the buffer until it is read in the program. So, if we were to accidentally send two positional values at one time (4 bytes of data), the data would not be lost, and this if statement would loop and until it read at least three of those bytes (until there is no longer more than one byte).

We can proceed in two ways from here. We can send the two individual bytes that represent the position to the two separate positional addresses of the motor (30 and 31 from Appendix 3), or we can combine the two numbers and just use the `SetPosition(id, pos)` function with the 2-byte number. The

second method is more efficient because it will send the data to the motor simultaneously, so we will use this way.

First, let us create variables to store these two values. Outside of both the setup and loop functions we can create two int variables. Let's call the first one high and the second one low (to represent the high and low byte values of our data). By convention I will initialize both of them to be -1. The code will look like this:



```

#include <ax12.h>

int high = -1;
int low = -1;

void setup()
{

```

Figure 6.3 - Variable Declaration and Initialization

Again, let's think of our limits and how they apply to the motor. We know we can send any value between 0 and 1023. In binary bytes, we can send between 00000000 0000 0000 (which is just 0), and 0000 0011 1111 1111 (which is 1023). It helps to look at our bytes individually. We will disregard the lower limit because it is trivial. If we look at the possible values of the high byte in the upper limit, we only have four possible values. In decimal, it is either 0, 1, 2, or 3. We can combine those with any range between 0 and 255 (the lower byte) to make up any value between 0 and 1023, but how? I highly encourage you to review this [page](#) before continuing.

After reviewing the binary to decimal conversion, we can take a look at how it applies to our data. Even though our number is being represented in two bytes, it is only a 10 bit number (the max value is 11 1111 1111, or 10 ones). Since we are splitting it into 2 bytes, we can look at our high byte value a little differently. Essentially, we can convert the binary value of the high byte to decimal, and just multiply it by 256, add it to the low byte, and get any range between 0 and 1023. An example is to convert 800 to binary. The high byte can be 0, 1, 2, or 3, and then we multiply by 256. We want the highest value we can get without going above our desired value. That would be $3 \times 256 = 768$. In binary, 3 is 11, so that is our high byte. Now, we have $800 - 768 = 32$ remaining to make up our number. 32 is 2^5 , so our binary value is 100000. Now, let's make up our bytes: the high byte is 0000 0011. The low byte is 0010 0000. Let's put them together ([concatenate](#)) and we get 0000 0011 0010 0000. You can manually convert or use an online converter, but $800 = 11\ 0010\ 0000$.

To quickly summarize the above, all we want to do is read in our high byte value, multiply it by 256 and add it to our low byte value. Now we have a design decision to make. Since the buffer behaves like a queue, we can either send the high byte first, or the low byte first. Since we normally think of appending the high byte to the front of the low byte, we will send the low byte first. This is important because we don't want to mistakenly multiply the low byte value by 256; that number would be way out of range!

To read data, we just use the `Serial.read()` function; this will read the piece of data "in the front of line," or whichever piece of data has been in the queue (buffer) the longest. As soon as we execute `Serial.read()`, that piece of data is removed from the queue and each piece of data shifts one spot in the

queue. So, let's read our first byte and save it as low, and then read the second byte to save it as high. Within the curly brackets that make up our if statement, it will look like this:

```
if(Serial.available() > 1)
{
    low = Serial.read();
    high = Serial.read();
}
```

Figure 6.4 - Reading Data from Serial Connection in Arduino

Now, all we have to do is use these numbers to set the position. Once we set the position, we will delay three seconds to allow the motor to get to its position, and then we will send those values back to MATLAB.

To write data back through the serial connection, we just use the `Serial.write` function (as we have done in the past). The difference now is that when MATLAB is connected the serial port, our Arduino serial monitor will not function anymore. We will have to read the values in MATLAB. Again, since we can only send 8 bits of data a time we are going to have to send the low byte and the high byte separately. We can just use two separate lines to do that using our `ax12GetRegister` function. Again, we will send the low byte first, followed by the high byte.

The final Arduino sketch will look like the following:

```

MatlabControl
#include <ax12.h>

int high = -1;
int low = -1;

void setup()
{
  //Initialize serial communication with a baud rate of 9600
  Serial.begin(9600);
}

void loop()
{
  if(Serial.available() > 1)
  {
    low = Serial.read();
    high = Serial.read();

    SetPosition(1, low + high*256);
    delay(3000);
    Serial.write(ax12GetRegister(1, 30, 1));
    Serial.write(ax12GetRegister(1, 31, 1));
  }
}

```

Figure 6.5 - MatlabControl Sketch

Now that our sketch is complete, upload it to the controller.

6.2.3 Sending Serial Data over MATLAB

Open MATLAB and create a new function. Call the function positionSet and have it take in two parameters: serial and goalPos. serial will be the serial port related to our connection, and goalPos will be our goal position (from 0 to 1023). Have it return one parameter, the position. This will look like the following:

```

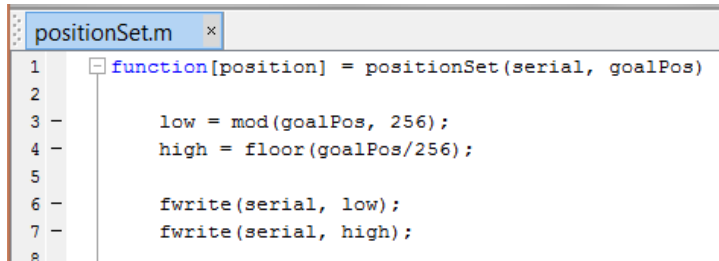
positionSet.m
1 function [position] = positionSet(serial, goalPos)
2

```

Figure 6.6 - positionSet Definition

To write data to a serial object, use the fwrite function with two parameters: the serial object and the data to send. In this case it would be: fwrite(serial, 255) to write 255 to the serial object. To read data from the serial, you use the fread function with the same serial object parameter; the second parameter in fread is the number of bytes to read. If we wanted to read in two bytes of data it would look something like this: fread(serial, 2).

So now we want to take our goalPos, break it up into a high bit and low bit value, and write them both to the serial. The code for this looks like the following:



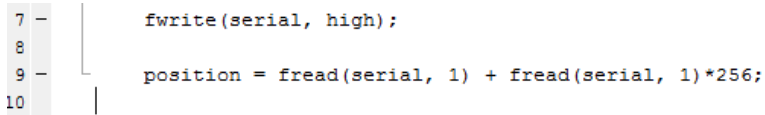
```

1 function[position] = positionSet(serial, goalPos)
2
3     low = mod(goalPos, 256);
4     high = floor(goalPos/256);
5
6     fwrite(serial, low);
7     fwrite(serial, high);
8

```

Figure 6.7 - Writing Data to Controller in MATLAB

The low byte takes the modulus of the goal position by 256 (which is the remainder value), and the high byte takes value of the goal position divided by 256, and rounds down. This is the mathematical representation of the logic we described above. We then write the two values to the board, with the low byte being written first, followed by the high byte. All that is left to do now is to set our return variable (position) equal to the values that the board is writing back to MATLAB. We can do this in one line as follows:



```

7     fwrite(serial, high);
8
9     position = fread(serial, 1) + fread(serial, 1)*256;
10

```

Figure 6.8 - Reading from Serial Connection in MATLAB

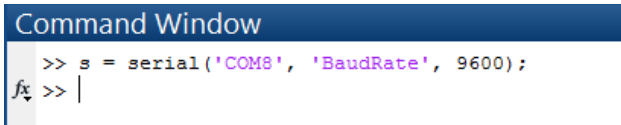
Since the data is stored on a buffer and is immediately removed once it is read, we can read the first value (which is the low byte), and then add it to the second read value (the high byte) * 256. So even though the `fread(serial,1)` function is called in the same manner two times in one line, they will actually have two separate values based on the buffer.

This completes our MATLAB function, and we can now use it once we define a serial object.

For now we will define our serial object in the MATLAB command window. If you were to automate this in a function, it would be a good idea to have a separate function to define the serial object and open the port as it takes a few seconds for the port to connect. You do not want to try writing to the controller immediately after opening the serial port; the data may not be transmitted properly.

The serial object in MATLAB can be initialized with a simple constructor, all it requires is the name of the port. This can be found in the Arduino IDE under "Tools" -> "Serial Port." Whichever one is selected when you have been uploading your sketches is the port you will use now. My port is "COM8," and will be referred to as such from now on.

Let's call our serial port object `s` and declare it in the MATLAB command window. We are going to use an overloaded constructor to also set the baud rate to the same rate as defined in the Arduino sketch (9600). This line looks like the following:



```

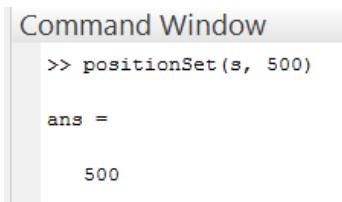
Command Window
>> s = serial('COM8', 'BaudRate', 9600);
fx >> |

```

Figure 6.9 - Creating a Serial Object in MATLAB

Note: Your implementation will look different depending on the name of your serial port. Once you have created the object, you can view more information about it by typing “s” and hitting enter (without a semicolon), or by typing “get(s)” (and again, without a semicolon). Many of the items are not super important for our current application, but you can find more about it on the web. The relevant properties for us are baud rate, and data bits. Data bits is the number of bits that it can send in each fwrite call; MATLAB defaults to 8 (one byte) and this is useful for us because we have always been talking in terms of bytes.

Now that the object is created, we want to open the port. To do so, type fopen(s); and hit enter. If you look at your ArbotiX controller, you should see the user LED flashing as it is opening. Once our serial object is set up and open, we can now use our positionSet function. The syntax is simple; type positionSet(s, [some value between 0 and 1023]) and our motor should move to that position. You can do this in the Command Window or write other MATLAB scripts or functions to include it. Recall that this function does have a return value, so you can save the position in a variable, or just view the answer by not storing it. The syntax is shown below:



```

Command Window
>> positionSet(s, 500)

ans =

    500

```

Figure 6.10 - Controlling Dynamixel Motor with MATLAB

Notice the delay between running the function and the answer. That is because we put a delay in our Arduino sketch so the motor arrives to its position before writing data back to the serial port. You may want to expand on the Arduino sketch to see how small of a delay can be used while still obtaining the desired result, or to monitor the motor’s rotation and only proceed once the desired angle is achieved.

When you are done controlling your motor using serial, it is very important to **close the serial port**. It is really easy to do and only takes a second, and you will encounter errors if you do not close the object. To do so, simply type fclose([serial object]); in our case it will be fclose(s). The object still exists, but the port is closed and can be used for other things (such as uploading new sketches). You will not be able to access the Arduino serial monitor while your port is open in MATLAB, and you will not be able to upload any new sketches.

It is also sometimes helpful to delete the object when you are all done so you do not accidentally create duplicate objects for the same port. To do this, simply type delete([object]); or in our case: delete(s). Again, this is why I chose to create our serial object outside of our function. When coding more complex

serial controllers, I highly recommend creating a function to create and open the port (you may need to add a pause after opening the port and sending commands) and then another function to close the port. That way you can do whatever you want with the port open in all of the other functions, but you will not leave the port open when you are done.

6.3 Serial Connection Summary

In summary, the important things to remember when controlling the motors using serial connections are:

- Define the objective as specifically and clearly as possible
 - Include things such as which data will be written in which order, how much data will I need before I can accomplish my goal, what data do I want to write back, and other such tasks
- Write the Arduino sketch to accomplish your goal with the received data
- Center your MATLAB function around the Arduino sketch so the relevant data is being sent

Chapter 7: Real Time Dynamixel Control Using C++

Note: It is recommended to have more advanced object oriented programing experience to utilize serial communication in C++.

We just learned how to do real time Dynamixel motor control through MATLAB's serial objects; however, sometimes it is more practical to extend other programs to write directly to the robocontroller. Many times, C++ will be a more useful language for programs like this. One example would be extending programs such as [AnimatLab](#) to allow for real time robotic control, rather than virtual simulations. This chapter will go through the basics of using C++ for serial control. It will utilize C++ Windows Form Applications to go through a visual example of how it works, but these concepts can all be extended to direct serial communication using the [System.IO.Ports::SerialPort](#) object. This object has many similar properties and methods as the object in MATLAB; however, in C++ the data must be casted appropriately to make sure the data being received by the controller is interpreted the way the user wants it to be interpreted.

7.1 MATLAB vs. C++

The biggest difference when using serial ports in MATLAB as opposed to C++ is how we represent our data that we want to send. In MATLAB it is easy; the data types are not casted so we can simply write an integer value to the controller and it will interpret it properly. In C++, it is not so simple. We will need to create an unsigned (since we are dealing with positive numbers) character array. We can then put our data into the array and write the values in the array to the controller.

This is really the biggest difference between MATLAB and C++ serial communication; the other components remain similar. You still need to upload an Arduino sketch to receive and interpret the data to make it do what you want to do, and you still need to send data one byte at a time.

7.2 The Arduino "Receiving" Sketch

As we learned in Chapter 6, to utilize serial communication with the Arduino robocontroller, we need to first upload a sketch that can interpret the data that is being received by the controller. For our example, we will use the same sketch that we used in chapter 6. All this sketch will do is receive data

through the serial port, and once two bytes have been received, it will set the motor's position accordingly. Right now upload the MatlabControl sketch we created in chapter 6 if it is not uploaded already.

7.3 C++ Windows Form Application for Serial Communication

For this example we will use a windows form application due to its simplicity. We will still be using the SerialPort object mentioned in the beginning of this chapter, and this object can be used in any other C++ program as well. For this example, we will be using Visual Studio 2010 on a PC running Windows.

7.3.1 Creating a C++ Windows Form Application for Serial Communication

To create a C++ Windows Form Application, open up Visual Studio 2010 and go to "File" -> "New" -> "Project". Under "Visual C++" go to "Windows Form Application" and name it "SerialTest." Your new project should have a blank form. Let's change the text of the form to "SerialTestForm" by right clicking on the form, clicking properties, and modifying the text property. Once you have modified the text of the form, your project should look like the following.

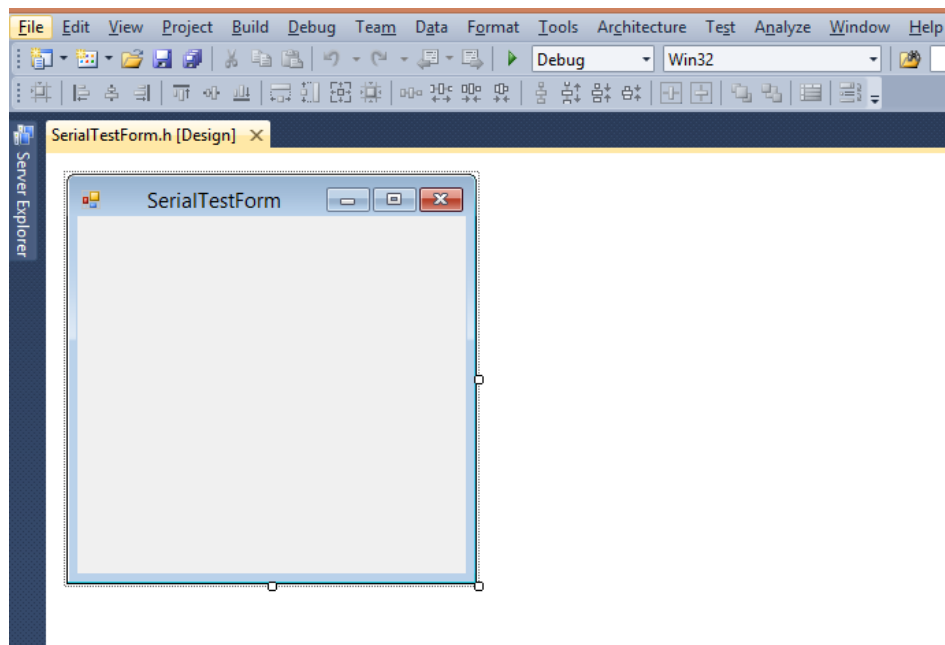


Figure 7.1 - Visual C++ Windows Form

The Windows Form Application automatically generates a lot of code behind the scenes that will display the form and initialize any controls on the form as soon as the program is built and run. We will not have to mess with any of the pre-generated code; we will only be working with adding new controls and modifying the code for said controls.

Now we are going to add items to our form to make a mini Serial test application. The first thing we will want to add to our form is a serial port object.

In order to add items to our Windows Form, we will need to open up the toolbox. Go to “View” -> “Toolbox” to open the toolbox. You can click on the thumbtack icon to dock it to one of the sides. Once you are viewing the toolbox, click on the “All Windows Form” carrot to view all windows form objects. When you do that, scroll down until you see “SerialPort” and click and drag that onto your Windows Form. What this will do is create a SerialPort object at the bottom of your form. Now we will want to modify the properties of the SerialPort object in our form; this is really easy to do when we use the Windows Form Application (which is why we are doing it in this example). To do so, double click on the SerialPort object that is at the bottom of Visual Studio. You should see the “Properties” menu in the bottom right of the screen; however, if this does not appear you may right click on the “SerialPort” object and then click “Properties.” The properties menu is shown below for reference.

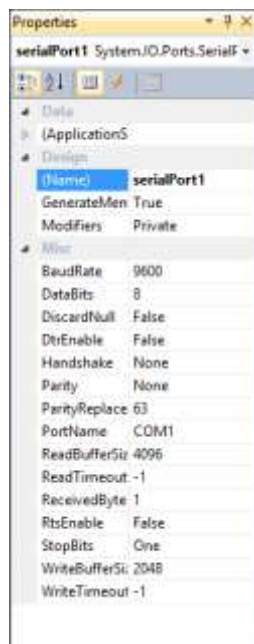


Figure 7.2 - SerialPort Properties

We want to make sure the BaudRate matches that of our Arduino sketch. In this case, the default is 9600 which we will use since it matches our Arduino sketch from chapter 6. We will also leave DataBits as 8 so we can write one byte of data at a time. The only property we will need to modify currently is the “PortName” property. We have been working with “COM8” for this text, so that is what we will give to our “PortName” property. **Note: This will be dependent upon your serial port; recall this can be checked in the Arduino IDE.**

Another property that may be modified as a good habit is the “WriteBufferSize.” The ArbotiX-M controller has a receiving buffer of 64 bytes, so we can change our “WriteBufferSize” to be 64 so we do not overflow the controllers receiving buffer. In this example it should not cause any issues, but it is a good habit to get into.

7.3.2 Adding Buttons and Other Controls to C++ Windows Form

Once our SerialPort object has been added to our form appropriately, we can now create a couple of buttons to run our tests. For now we will add three buttons; one to open the serial port, one to send a position value, and one to close the serial port. To add a button to the form, return to the toolbox and scroll until you find “Button.” Again, click and drag it onto the form. When you click on the button, you should see the button’s properties in the property menu. Let us change both the name of the button and the text of the button. We will make the name “openSerial” and we will change the text to say “Open Serial Port.” You may also have to resize the button to show all of the text.

Do the same thing for two more buttons, but make the name of the second one “changePosition” with button text “Change Position,” and make the third one with name “closeSerial” and with text “Close Serial Port.” Your form should look like the following.

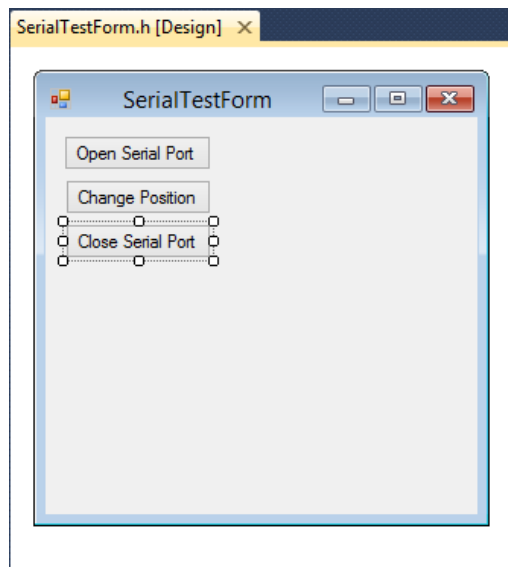


Figure 7.3 - Windows Form with Buttons

We are now ready to start putting some code behind our buttons.

7.3.3 Making Windows Forms Controls Carry out Functions

To add code to a button all you have to do is double-click on the button. When you double-click, Visual Studio will automatically take you to the method that will run when you click on the button during runtime. For now, let’s double-click on the “Open Serial Port” button and add the code to open the

serial port. The code to open the serial port is really simple; it is just this one line: “this->serialPort1->Open();”. In Visual Studio, it looks like the following:

```
#pragma endregion
private: System::Void openSerial_Click(System::Object^ sender, System::EventArgs^ e) {
    this->serialPort1->Open();
}
};
```

Figure 7.4 - Open Serial Port

We can repeat this process for the “Close Serial Port” button with the only difference being that we write “this->serialPort1->Close();” instead of Open().

Now that the code for our first two buttons is complete, we can do the more complicated button. When we click the “Change Position” button we will want it to send a positional value to the controller to be interpreted by the board. We can do this in two ways; we can code it to break it up into low and high byte values (as we did with the MATLAB code) or we can just have it take in two parameters (and low and high byte) and do it manually. We will have the code do it for us so we only need to enter one position value. First though, we will need to add a control to accept a value.

Return to the Form1.h[Design] tab in Visual studio and add a “Text Box” from the toolbox (where you found the buttons and the SerialPort object). Under its properties menu, give it the name “positionBox.” Also, add a label above the text box and give it a text of “Position Input:”.

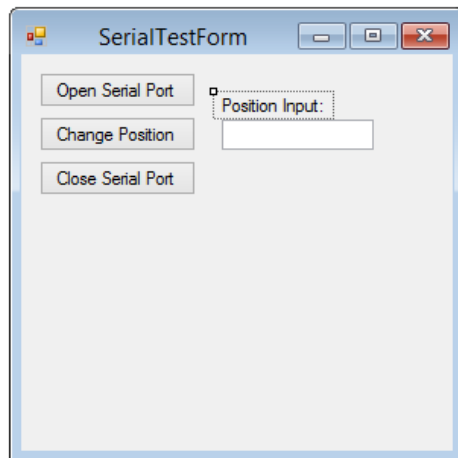


Figure 7.5 - Final SerialTestForm Appearance

Now, double click on the “Change Position” textbox and let’s add our code. **Note: All code added from now on in this chapter will be in the “private: System::Void changePosition_Click(...)” method.**

What we want it to do is read the value that will be entered in the “positionBox” textbox, break into a low and high byte, and send it to the controller. The first thing we will want to do though is create an array to store our data. Let’s call it “data” and it will have type unsigned char. We will want it to have length two (for both the low and the high byte). The constructor call will look like this:

```
array<unsigned char>^ data = gcnew array<unsigned char>(2);
```

The “^” means that this is a reference to a managed object that can be moved around in the memory. The pointer will follow the data as it is moved in the memory, and it will also be garbage collected automatically.

We will now need to take the value from the “positionBox” and break it into its low and high bytes and add it to the array. The low byte will go in the 0th position and the high byte will go in the 1st position because of the design of our Arduino sketch. Since we are using C++, we will have to be more careful with our data types. When you type anything into the “positionBox” textbox, it is automatically considered a `System::String` object; however, we need it to be an integer, so we will have to parse the string to convert it to an integer. For now, let us store our full integer (which may be more than one byte) into one variable, and we will convert to low bytes afterwards.

The line of code to create the integer, read the value from the textbox, and convert it to an integer is as follows:

```
_int32 position = System::Int32::Parse(this->positionBox->Text);
```

The “this->positionBox->Text” returns the value of the text in the box, and the `System::Int32::Parse` converts it to a 32 bit integer. A 32 bit integer is more than enough space for our two byte positional values. Now that we have our full position value stored as an integer, the rest is similar to MATLAB. We will have to take the modulus of the value by 256 and store it as the low byte (or the 0th position of the array) and then we will have to divide the value by 256 and store it as the high byte (or the 1st position of the array). We do not have to do any rounding when we divide since we are working with integer values and integers will always truncate the decimal regardless.

The two lines of code to do this are the following:

```
data[0] = position%256;
data[1] = position/256;
```

Now we have our data broken up into the two bytes that our Arduino sketch is prepared to interpret; all we have to do now is send it. We can use the “Write” method of the `Serial` object to send the data over the serial port. In chapter 6 we briefly mentioned overloaded methods; the “Write” method actually exists in three forms summarized on the `SerialPort` MSDN page and reproduced below.




	<code>Write(String)</code>	Writes the specified string to the serial port.
	<code>Write(array<Byte>, Int32, Int32)</code>	Writes a specified number of bytes to the serial port using data from a buffer.
	<code>Write(array<Char>, Int32, Int32)</code>	Writes a specified number of characters to the serial port using data from a buffer.

Figure 7.6 - Serial.Write Overloaded Methods

The most useful method for us is the third one; we can express integer values as character arrays. The second method can also be used if you convert all values down to their byte forms, but this requires extra steps. The first parameter of the third method is the data you want to write. In this case, we will write the “data” array. The second parameter is the offset in the buffer at which to begin copying bytes to the port. For our example it (and most applications) this will simply be 0. The third parameter is the count, or number of characters to write to the port. In this case we will think of each “character” as a full byte (so it could be a two or three digit integer). In this example, we are sending 2 bytes so our third parameter is 2. To write data to the serial port, the line of code will look like the following.

```
this->serialPort1->Write(data, 0, 2);
```

Now our code will be functional and we will be able to change the position of the motor by writing a value in the “positionBox” textbox and clicking the “Change Position” button. Recall, however, that our Arduino sketch also writes values back to the computer. We will want to read these values and do something with them.

Again, we will want to create an unsigned character array with memory for two values. Let’s call this array “read.” The constructor will be identical to that of “data” with the exception of “data” being replaced by “read.” We will now utilize the SerialPort’s “read” method. Again there are several methods for reading individual bytes, characters or arrays; refer to the [SerialPort MSDN](#) page for a full list. We will use the method that reads a number of characters from the buffer; in this case it will be two. The code will look like the following.

```
this->serialPort1->Read(read, 0, 2);
```

This will read two bytes from the buffer with no offset and store them in the “read” array we just created.

Now that we have our data read back to our computer we can do something with it. An exercise to try is to create two new textboxes that will display the low and the high byte. Give it a shot and look below to see the code if you are confused. Remember you will have to convert any value you want to display to a string. To convert an unsigned character to a string you can use the “System::Convert::ToString(…)” method. For my code, I left the textbox names as the default “textBox1” and “textBox2.” You will have to drag and drop from the toolbox as we did before.

The final code for the “changePosition_Click” method (which runs when you click the “Change Position” button) is listed below.

```
private: System::Void changePosition_Click(System::Object^ sender, System::EventArgs^ e) {
    array<unsigned char>^ data = gcnew array<unsigned char>(2);
    _int32 position = System::Int32::Parse(this->positionBox->Text);
    data[0] = position%256;
    data[1] = position/256;

    this->serialPort1->Write(data, 0, 2);

    array<unsigned char>^ read = gcnew array<unsigned char>(2);

    this->serialPort1->Read(read, 0, 2);

    this->textBox1->Text = System::Convert::ToString(read[0]);
    this->textBox2->Text = System::Convert::ToString(read[1]);
}
```

Figure 7.7 - changePosition_Click Method

Once you are done you may run the program by pressing “F5” or clicking the “Start Debugging” (looks like the play button) at the top of Visual Studio. **If you get an error** saying: “The system cannot find the file specified (in reference to a .exe)” or notice in the runtime box: “fatal error LNK1123: failure during conversion to COFF” this indicates that you need to install Service Pack 1 for Visual Studio 2010. It can be found [here](#). Once your application builds correctly, it can be run by clicking the “Open Serial Port” button, then enter in data into the textbox labeled “Position Input:” (remember to make sure the number is in-between 0 and 1023 for an AX-12A) and click “Change Position.” The values should appear in the two newly created text boxes. If it is not working properly, double check the properties of the serial port and the serial port connection to make sure everything matches.

To utilize the SerialPort class in a non-visual application (such as a console application), the example provided on the [MSDN](#) page at the bottom is extremely useful. It goes through a detailed console application of the SerialPort class.

7.4 Other Serial Resources

We just learned how to control our Dynamixel motor in real-time using a serial connection and MATLAB as well as C++. Other resources exist to send and receive data over serial ports. MATLAB just happens to be one of the easiest to implement and C++ is one of the most widely used. All of the basics regarding serial communication and serial objects will hold true for most languages. Many libraries exist out there to help you read data from and write data to serial connections. I’ve listed several resources I’ve encountered that may help you with your applications. These are more advanced topics that are not entirely covered in the scope of this text.

[Serial Library for C++](#)

[Java Simple Serial Connector](#)

[Using Arduino to Create Virtual Serial Ports \(Useful for Multiple Controllers\)](#)

Appendix 1: Sources

In order of appearance:

[Wikipedia](#)

[Trossen Robotics](#)

[Arduino](#)

[Robotis](#)

[wikiHow](#)

[Google Code](#)

[Code Project](#)

[MSDN](#)

Appendix 2: Additional Resources

Appendix 2.1 Manually Changing Dynamixel ID

If you connect a Dynamixel motor to your controller and cannot get a response, it may be necessary to reset the ID of the motor. There is a piece of software produced by TrossenRobotics that allows us to visually reset the ID of a motor. It is called DynaManager and can be downloaded at the [link](#) provided. Full instructions to resetting the ID of a Dynamixel motor can be found on TrossenRobotics' website. The instructions are found [here](#). You **must** have the ros sketch installed on your ArbotiX controller for the DynaManager software to work properly.

Appendix 2.2 Restoring Dynamixel to Factory Defaults

If you use the DynaManager software and encounter an error that says the Dynamixel motor cannot be found, a firmware reset may be necessary. For the firmware reset, you will need an [USB2Dynamixel](#) connector. You will also need the [RoboPlus](#) software. Make sure to install the latest version. With RoboPlus installed and the Dynamixel motor you want to reset connected to the USB2Dynamixel connector, follow the instructions found [here](#). A good way to check if the firmware reset fixed your motor is to follow the instructions found in Appendix 2.1 to manually set the ID of the motor. If the DynaManager software recognizes the motor, it should work properly with your sketches on the ArbotiX controller.

Appendix 3: AX-12 Property Address Table

Each property name in the table will link to the specifics about that property, including the high and low limits, as well as what it does. **Note:** This table is specific to AX-12 motors; for other series you can find equivalent tables from the following page: [Robotis Dynamixel Manuals](#). Just navigate to the series of interest.

The way to read this table is as follows:

- Identify the property of interest in which you will need to read data from or write data to
- Find it in the "Name" column of the table
- Look at the address in which it is located
 - For properties that have two addresses, the first address will always be the low bit and the second address will always be the high bit

- More information on this can be found in section 4.2
- To determine the limits of the property, as well as the units that map to the value you are sending, click on the link in the “Name” column
- You now know what address to send data to or read data from, and you know the upper and lower limits of the data it can physically accept

A few other things to be aware of: the “Access” column identifies if you can read data from that address (R), write data to that address (W), or both (RW). The “Initial Value” column is the default value of that address out of the box. If your motor is acting strange and you have to restore it to factory defaults, these are the values that will be sent to each address. The “Area” column signifies if the address lies in the [EEPROM](#) or the [RAM](#). EEPROM and RAM are both types of memory, the major difference being that the values stored in the addresses that lie within the EEPROM will be saved when power is removed, while the RAM values will be reset. So if I change the ID of a Dynamixel motor, it will save it until I change it again (since it lies in the EEPROM).

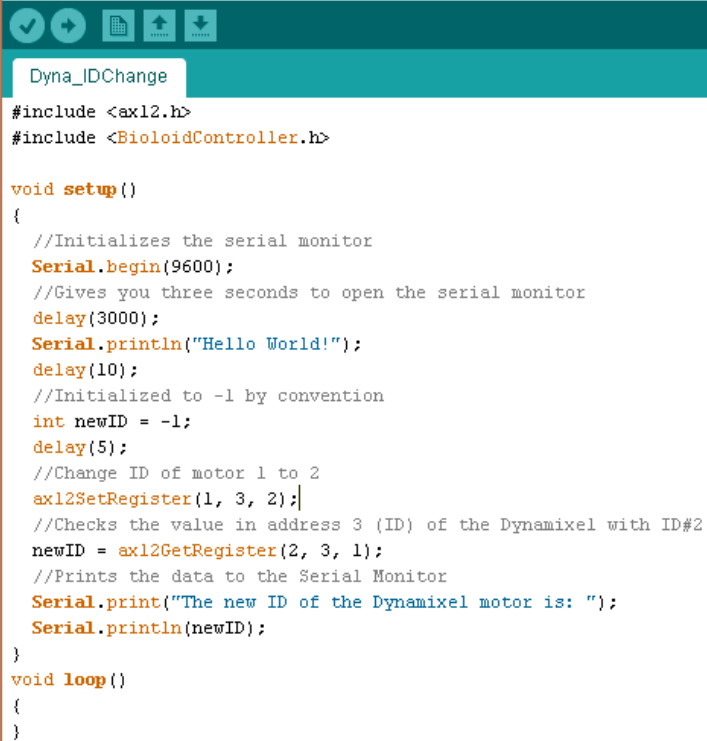
Area	Address (Hexadecimal)	Name	Description	Access	Initial Value (Hexadecimal)
EEPROM	0 (0X00)	Model Number(L)	Lowest byte of model number	R	12 (0X0C)
	1 (0X01)	Model Number(H)	Highest byte of model number	R	0 (0X00)
	2 (0X02)	Version of Firmware	Information on the version of firmware	R	-
	3 (0X03)	ID	ID of Dynamixel	RW	1 (0X01)
	4 (0X04)	Baud Rate	Baud Rate of Dynamixel	RW	1 (0X01)
	5 (0X05)	Return Delay Time	Return Delay Time	RW	250 (0XFA)
	6 (0X06)	CW Angle Limit(L)	Lowest byte of clockwise Angle Limit	RW	0 (0X00)
	7 (0X07)	CW Angle Limit(H)	Highest byte of clockwise Angle Limit	RW	0 (0X00)
	8 (0X08)	CCW Angle Limit(L)	Lowest byte of counterclockwise Angle Limit	RW	255 (0XFF)
	9 (0X09)	CCW Angle Limit(H)	Highest byte of counterclockwise Angle Limit	RW	3 (0X03)
	11 (0X0B)	the Highest Limit Temperature	Internal Limit Temperature	RW	70 (0X46)
	12 (0X0C)	the Lowest Limit Voltage	Lowest Limit Voltage	RW	60 (0X3C)
	13 (0X0D)	the Highest Limit Voltage	Highest Limit Voltage	RW	140 (0XBE)
	14 (0X0E)	Max Torque(L)	Lowest byte of Max. Torque	RW	255 (0XFF)
	15 (0X0F)	Max Torque(H)	Highest byte of Max. Torque	RW	3 (0X03)
	16 (0X10)	Status Return Level	Status Return Level	RW	2 (0X02)
	17 (0X11)	Alarm LED	LED for Alarm	RW	36(0x24)
	18 (0X12)	Alarm Shutdown	Shutdown for Alarm	RW	36(0x24)

R A M	24 (0X18)	Torque Enable	Torque On/Off	RW	0 (0X00)
	25 (0X19)	LED	LED On/Off	RW	0 (0X00)
	26 (0X1A)	CW Compliance Margin	CW Compliance margin	RW	1 (0X01)
	27 (0X1B)	CCW Compliance Margin	CCW Compliance margin	RW	1 (0X01)
	28 (0X1C)	CW Compliance Slope	CW Compliance slope	RW	32 (0X20)
	29 (0X1D)	CCW Compliance Slope	CCW Compliance slope	RW	32 (0X20)
	30 (0X1E)	Goal Position(L)	Lowest byte of Goal Position	RW	-
	31 (0X1F)	Goal Position(H)	Highest byte of Goal Position	RW	-
	32 (0X20)	Moving Speed(L)	Lowest byte of Moving Speed	RW	-
	33 (0X21)	Moving Speed(H)	Highest byte of Moving Speed	RW	-
	34 (0X22)	Torque Limit(L)	Lowest byte of Torque Limit	RW	ADD14
	35 (0X23)	Torque Limit(H)	Highest byte of Torque Limit	RW	ADD15
	36 (0X24)	Present Position(L)	Lowest byte of Current Position	R	-
	37 (0X25)	Present Position(H)	Highest byte of Current Position	R	-
	38 (0X26)	Present Speed(L)	Lowest byte of Current Speed	R	-
	39 (0X27)	Present Speed(H)	Highest byte of Current Speed	R	-
	40 (0X28)	Present Load(L)	Lowest byte of Current Load	R	-
	41 (0X29)	Present Load(H)	Highest byte of Current Load	R	-
	42 (0X2A)	Present Voltage	Current Voltage	R	-
	43 (0X2B)	Present Temperature	Current Temperature	R	-
	44 (0X2C)	Registered	Means if Instruction is registered	R	0 (0X00)
	46 (0X2E)	Moving	Means if there is any movement	R	0 (0X00)
	47 (0X2F)	Lock	Locking EEPROM	RW	0 (0X00)
	48 (0X30)	Punch(L)	Lowest byte of Punch	RW	32 (0X20)
	49 (0X31)	Punch(H)	Highest byte of Punch	RW	0 (0X00)

Appendix 4: Code Base for Examples in This Text

Appendix 4.1 Arduino Sketches

Appendix 4.1.1 Dyna_IDChange


The image shows a screenshot of an Arduino IDE window. The title bar at the top reads "Dyna_IDChange". The code area contains the following C++ code:

```
#include <ax12.h>
#include <BioloidController.h>

void setup()
{
  //Initializes the serial monitor
  Serial.begin(9600);
  //Gives you three seconds to open the serial monitor
  delay(3000);
  Serial.println("Hello World!");
  delay(10);
  //Initialized to -1 by convention
  int newID = -1;
  delay(5);
  //Change ID of motor 1 to 2
  ax12SetRegister(1, 3, 2);
  //Checks the value in address 3 (ID) of the Dynamixel with ID#2
  newID = ax12GetRegister(2, 3, 1);
  //Prints the data to the Serial Monitor
  Serial.print("The new ID of the Dynamixel motor is: ");
  Serial.println(newID);
}

void loop()
{
}
```

Appendix 4.1.2 Dyna_PositionSet



```

#include <ax12.h>
#include <BioloidController.h>

void setup() {
  //Sets the position of motor with ID#1 to 512
  SetPosition(1, 512);
  delay(3000);
}

void loop() {
  SetPosition(1, 0);
  delay(3000);
  SetPosition(1, 1023);
  delay(3000);
}

```

Appendix 4.1.3 Dyna_PositionSet2



```


#include <ax12.h>
#include <BioloidController.h>

void setup()
{
  //Sets the position of motor with ID#1 to 512
  ax12SetRegister2(1, 30, 512);
  delay(3000);
}

void loop()
{
  ax12SetRegister2(1, 30, 0);
  delay(3000);
  ax12SetRegister2(1, 30, 1023);
  delay(3000);
}

```

Appendix 4.1.4 DynaTorqueTesting




DynaTorqueTesting

```
#include <DynamixelTorqueControl.h>
#include <ax12.h>

void setup()
{
    applyTorque(1, 0, 500);
}
void loop()
{
}
```

Appendix 4.1.5 MatlabControl



MatlabControl

```
#include <ax12.h>

int high = -1;
int low = -1;


void setup()
{
    //Initialize serial communication with a baud rate of 9600
    Serial.begin(9600);
}

void loop()
{
    if(Serial.available() > 1)
    {
        low = Serial.read();
        high = Serial.read();

        SetPosition(1, low + high*256);
        delay(3000);

        Serial.write(ax12GetRegister(1, 30, 1));
        Serial.write(ax12GetRegister(1, 31, 1));
    }
}
```


Appendix 4.1.6 FunctionTest



```

FunctionTest$
#include <axl2.h>

void setup()
{
    //Sets the robot to its rest position
    setRestPosition();
    //Adjusts position of motor 2 based on position of motor 3
    int pos2 = adjustPosition(3, 2);
}

void loop()
{
}

//This function will return the robot to its rest position
void setRestPosition()
{
    SetPosition(1, 100);
    SetPosition(2, 200);
    SetPosition(3, 300);
    SetPosition(4, 400);
    SetPosition(5, 500);
    SetPosition(6, 600);
    delay(3000);
}

//Set position of motor b based on position of motor a
//Returns the new value of the position of motor b
int adjustPosition(int a, int b)
{
    //Gets two bytes of data starting at address 36 of motor A
    int aPos = axl2GetRegister(a, 36, 2);

    if(aPos < 512)
    {
        SetPosition(b, aPos);
        return aPos;
    }

    else
    {
        SetPosition(b, 1023);
        return 1023;
    }
}

```

Appendix 4.2 C++ Library Code

Appendix 4.2.1 Header File

```
DynamixelTorqueControl.h  X DynamixelTorqueControl.cpp
1  #ifndef DynamixelTorqueControl_h
2  #define DynamixelTorqueControl_h
3
4  #include "ax12.h"
5
6  void applyTorque(int id, int direction, int magnitude);
7
8  #endif
```

Appendix 4.2.2 C++ File

```
DynamixelTorqueControl.h  DynamixelTorqueControl.cpp  X
1  #include "DynamixelTorqueControl.h"
2  #include "ax12.h"
3
4  void applyTorque(int id, int direction, int magnitude)
5  {
6      ax12SetRegister2(id, AX_TORQUE_LIMIT_L, magnitude);
7      SetPosition(id, direction*1023);
8  }
```

Appendix 4.3 MATLAB Serial Control Code

Appendix 4.3.1 Serial Object Definition

Command Window

```
>> s = serial('COM8', 'BaudRate', 9600)
```

```
Serial Port Object : Serial-COM8
```

Communication Settings

```
Port:      COM8
BaudRate:   9600
Terminator: 'LF'
```

Communication State

```
Status:     closed
RecordStatus: off
```

Read/Write State

```
TransferStatus: idle
BytesAvailable: 0
ValuesReceived: 0
ValuesSent:     0
```

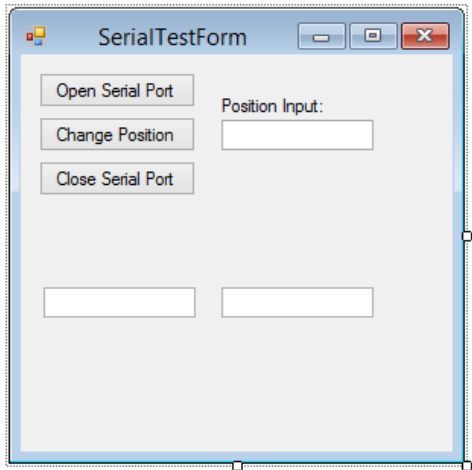
Appendix 4.3.2 positionSet Function

```
positionSet.m x
1 function[position] = positionSet(serial, goalPos)
2
3     low = mod(goalPos, 256);
4     high = floor(goalPos/256);
5
6     fwrite(serial, low);
7     fwrite(serial, high);
8
9     position = fread(serial, 1) + fread(serial, 1)*256;
10
```

Appendix 4.4 C++ Serial Control Code

The pre-generated code is not included in this appendix; only the code written for the example in Chapter 7 is displayed.

Appendix 4.4.1 Form Layout



Appendix 4.4.2 openSerial_Click Method

```
private: System::Void openSerial_Click(System::Object^ sender, System::EventArgs^ e) {
    this->serialPort1->Open();
}
```

Appendix 4.4.3 closeSerial_Click Method

```
private: System::Void closeSerial_Click(System::Object^ sender, System::EventArgs^ e) {
    this->serialPort1->Close();
}
```

Appendix 4.4.4 changePosition_Click Method

```
private: System::Void changePosition_Click(System::Object^ sender, System::EventArgs^ e) {
    array<unsigned char>^ data = gcnew array<unsigned char>(2);
    _int32 position = System::Int32::Parse(this->positionBox->Text);
    data[0] = position%256;
    data[1] = position/256;

    this->serialPort1->Write(data, 0, 2);

    array<unsigned char>^ read = gcnew array<unsigned char>(2);

    this->serialPort1->Read(read, 0, 2);

    this->textBox1->Text = System::Convert::ToString(read[0]);
    this->textBox2->Text = System::Convert::ToString(read[1]);
}
```