**Hands-on Lab**

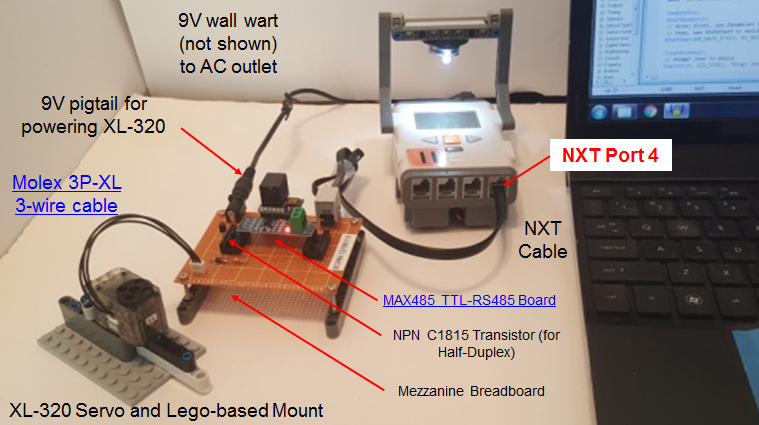
**XL-320 NXC Programming – “Hello World (LED)” Example**

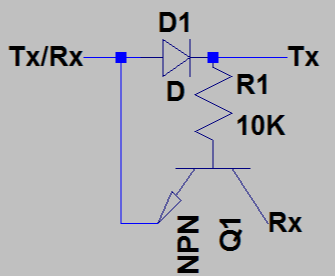
This lab introduces NXC Programming of the Robotis XL-320 Dynamixel servo. The Lego NXT Brick’s Port 4 features a serial interface (RS-485 protocol). This allows the Brick to communicate to the TTL-level serial port on the XL-320. Changing the XL-320’s LED color is a “Hello World” example to introduce RS-485 programming and writing instructions to the XL-320.

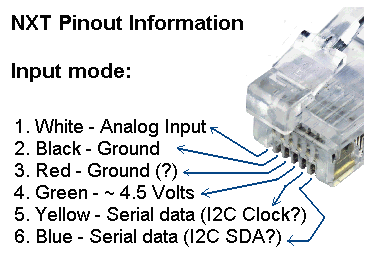
**Preliminary:** Hardware connections and explanation

**Hardware Connections**

**Figure A:** NXT-to-XL320 connections (left). NPN Transistor for half-duplex (top right) and NXT cable wire description (bottom right)







**Figure A** shows the hardware connections. RS-485 protocol digitizes at -7 to +12 Volts. However, the XL-320 uses transistor-to-transistor logic (TTL) to convert bytes digitally (+5V and Ground). As such, a converter is needed. The Maxim MAX485 is a popular chip for such conversion. Its popularity is underscored by $2 boards complete with supporting passive components. One caveat of the XL-320 is that it uses half-duplex RS-485 interfacing. The Molex connector has wires for power, ground, and data. Thus only 1-wire is used to read and/or write bytes (i.e. half-duplex). As such, a NPN transistor is used to implement half-duplexing (**Figure A top right**). Lastly, the NXT cable’s Yellow (YLW) and Blue (BLU) wires (**Figure A bottom right**) serve serial purposes when RS485 is invoked.

**Dynamixel Protocol 2.0 and XL-320 Firmware**

As introduced earlier, the XL-320 is a *smart* servo; firmware is embedded. Firmware is used to permanently hold device information like identifiers (e.g. model or ID number, communication settings like baud rate, and firmware version). Firmware often consists of Random Access Memory (*RAM*) and/or Electronically Erasable Programmable Read-Only Memory (*EEPROM*). RAM holds temporary information like encoder positions and LED states. EEPROM stores more permanent information like read/write instructions.

Robotis’ information on the XL-320 is comprehensive, albeit cryptic:

1. XL-320 specifications <http://emanual.robotis.com/docs/en/dxl/x/xl320/>
2. EEPROM and RAM Control Table <http://emanual.robotis.com/docs/en/dxl/x/xl320/#control-table>
3. Robotis Protocol 2.0 Instruction and Status packets and Packet Processing <http://emanual.robotis.com/docs/en/dxl/protocol2/>

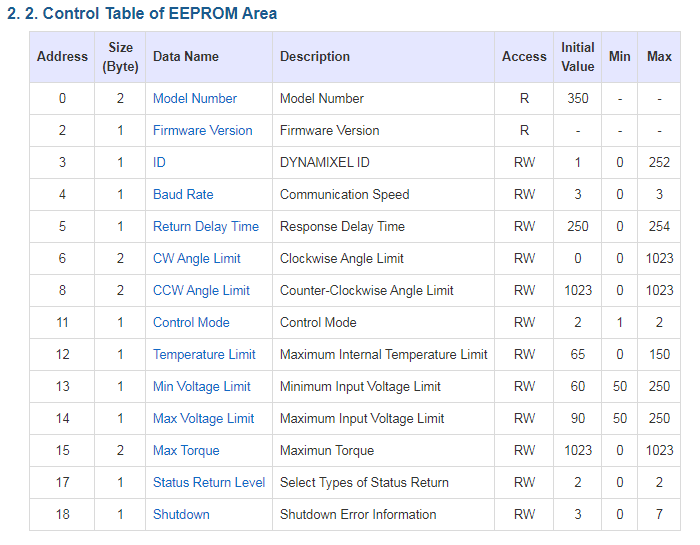
The XL-320 has powerful features and the above links are needed to exploit them.

**Concept 1 Create Definition Header File (H-File) xl320-defines1\_0a.h**

**Step 1:** Create Definition Header File (H-File) – EEPROM Area

A cursory view of links underscores many details of the XL-320. It is helpful to first create a definition header file. This file will #define constants that will be needed to read and/or write Instruction and Status packets. Such packets reference the firmware to command the XL-320.

Section 2.2 of <http://emanual.robotis.com/docs/en/dxl/x/xl320/> details the EEPROM Control Table and shown in **Figure 1A**.



**Figure 1A:** Addresses (in Decimal) for each Data Name in **EEPROM**. This table can be found in [Section 2.2](http://emanual.robotis.com/docs/en/dxl/x/xl320/#control-table) (Control Table) of the Robotis XL-320 E-Manual.

In the C programming language, all-uppercase is conventionally used to define constants. For this and future labs, the prefix EEPROM\_, RAM\_, and INSTRUCTION\_ will be used before the Data Name. Also, the underscore character will be used between each word. Example, for Model Number (top line in **Figure 1A**) would be represented in the H-file as:

#define EEPROM\_MODEL\_NUMBER 0x00 // 2 bytes; motor’s model number

Following this conventional, the EEPROM defines for the H-file (Yellow highlight) would look like **Figure 1B**.

// FILE: xl320-defines1\_0a.h

// AUTH: P.Oh

// DATE: 09/19/19 12:24

// VERS: 1.0a: XL-320 motor defines in Control Table; no functions in this file

// DESC: Refers to Section 2

// http://emanual.robotis.com/docs/en/dxl/x/xl320/#control-table

// Section 2.2 is EEPROM Control Table defines

// Section 2.3 is the RAM Control Table defines

// REFS: F:\nationalInstruments\nxcProjects\rs-485\dynamixel\Dynamixel XL-320\

// paulOhDynamixelXl320HeaderFile-1.0d.h

// Instruction related Defines

#define HEADER\_1 0xFF // For Instruction Packet Header 1

#define HEADER\_2 0xFF // For Instruction Packet Header 2

#define HEADER\_3 0xFD // For Instruction Packet Header 3

#define RESERVED 0x00 // For Instruction Packet Reserved

// EEPROM Address related Defines

// See Robotis Section 2.2 http://emanual.robotis.com/docs/en/dxl/x/xl320/

#define EEPROM\_MODEL\_NUMBER 0x00 // 2 bytes; motor's model number

#define EEPROM\_FIRMWARE\_VERSION 0x02 // 1 byte; motor's firmware version

#define EEPROM\_ID 0x03 // 1 byte; motor's ID number [0-252]

#define EEPROM\_BAUD\_RATE 0x04 // 1 byte; baud [0-3]

#define EEPROM\_RETURN\_DELAY\_TIME 0x05 // 1 byte; instruction packet send time

#define EEPROM\_CW\_ANGLE\_LIMIT 0x06 // 2 bytes; minimum value of Goal Position

#define EEPROM\_CCW\_ANGLE\_LIMIT 0X08 // 2 bytes; maximum value of Goal Position

#define EEPROM\_CONTROL\_MODE 0x0B // 1 byte; Wheel (1) or Joint (2) modes

#define EEPROM\_TEMPERATURE\_LIMIT 0x0C // 1 byte; overheat shutdown value [0-100]

#define EEPROM\_MIN\_VOLTAGE\_LIMIT 0x0D // 1 byte; minimum operational voltage

#define EEPROM\_MAX\_VOLTAGE\_LIMIT 0x0E // 1 byte; maximum operational voltage

#define EEPROM\_MAX\_TORQUE 0x0F // 2 bytes; maximum torque value

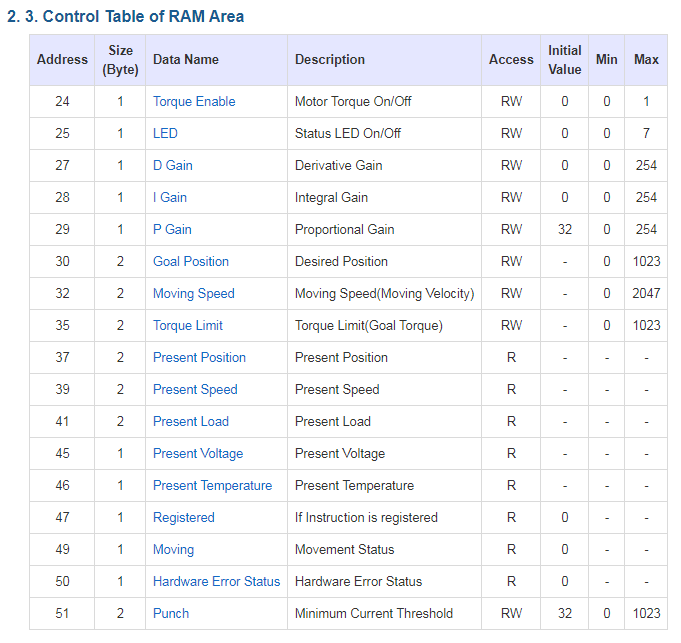
#define EEPROM\_STATUS\_RETURN\_LEVEL 0x11 // 1 byte; how to send status packet

#define EEPROM\_SHUTDOWN 0x12 // 1 byte; when to shutdown motor

**Figure 1B**: #defines for the **EEPROM** Control Table constants

**Step 2:** Create Definition Header File (H-File) – RAM Area

Section 2.3 of <http://emanual.robotis.com/docs/en/dxl/x/xl320/> details the RAM Control Table and shown in **Figure 1C**. Following the aforementioned naming convention, **Figure 1D** shows the #defines to be added to the H-file in **Figure 1B**.



**Figure 1C:** Addresses (in Decimal) for each Data Name in **RAM**. This table can be found in [Section 2.2](http://emanual.robotis.com/docs/en/dxl/x/xl320/#control-table) (Control Table) of the Robotis XL-320 E-Manual.

// RAM Address related Defines

// See Robotis Section 2.3 http://emanual.robotis.com/docs/en/dxl/x/xl320/

#define RAM\_TORQUE\_ENABLE 0x18 // 1 byte; turns on/off torque control

#define RAM\_LED 0x19 // 1 byte; changes motor's LED color

#define RAM\_D\_GAIN 0x1B // 1 byte; motor's derivative gain

#define RAM\_I\_GAIN 0x1C // 1 byte; motor's integral gain

#define RAM\_P\_GAIN 0x1D // 1 byte; motor's proportional gain

#define RAM\_GOAL\_POSITION 0x1E // 2 bytes; destination position value

#define RAM\_MOVING\_SPEED 0x20 // 2 bytes; Wheel or Joint dependent

#define RAM\_TORQUE\_LIMIT 0x23 // 2 bytes; maximum torque limit value

#define RAM\_PRESENT\_POSITION 0x25 // 2 bytes; motor's present position

#define RAM\_PRESENT\_SPEED 0x27 // 2 bytes; Wheel or Joint mode dependent [0-2047]

#define RAM\_PRESENT\_LOAD 0x29 // 2 bytes; currently applied load value is [0-2047]

#define RAM\_PRESENT\_VOLTAGE 0x2D // 1 byte; present supply voltage

#define RAM\_PRESENT\_TEMPERATURE 0x2E // 1 byte; motor's internal temperature in Celsius

#define RAM\_REGISTERED 0x2F // 1 byte; REG\_WRITE instruction received or not

#define RAM\_MOVING 0x31 // 1 byte; Goal Position completed or in-progress

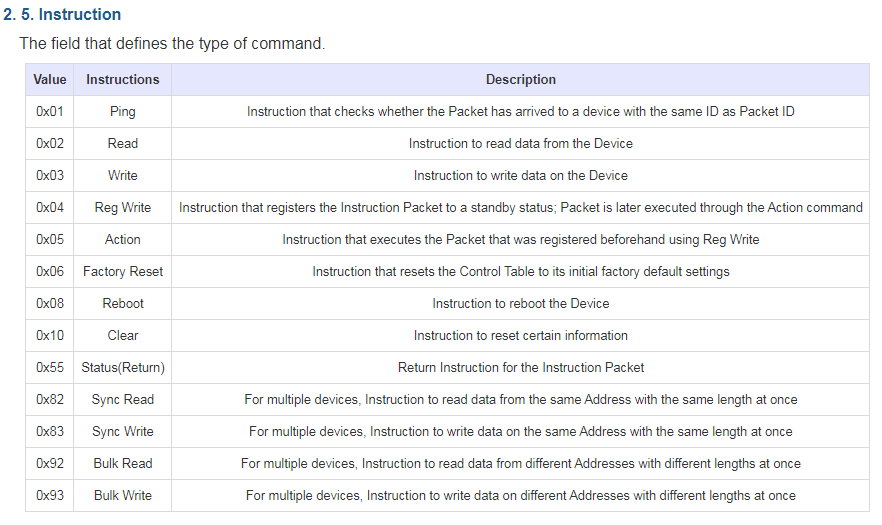
#define RAM\_HARDWARE\_ERROR\_STATUS 0x32 // 1 byte; present hardware error status

#define RAM\_PUNCH 0x33 // 2 bytes; minimum current to drive motor

**Figure 1D**: #defines for the **RAM** Control Table constants

**Step 3:** Create Definition Header File (H-File) – Instruction Area

The XL-320 is one of about a dozen different Dynamixel servos. Their EEPROM and RAM constants may differ but instructions are common. Robotis calls this Protocol 2.0. Section 2.5 of <http://emanual.robotis.com/docs/en/dxl/protocol2/#instruction-packet> details the Instruction Packet and given in **Figure 1E**.



**Figure 1E:** Addresses (in Hex) for each Instruction. This table can be found in [Section 2.5](http://emanual.robotis.com/docs/en/dxl/protocol2/#instruction-packet) (Instruction) of the Robotis Protocol 2.0 E-Manual.

Continuing with the naming convention, the #defines in **Figure 1F** can be added to the H-file.

// Instruction related Defines

// See Section 2.5

// http://emanual.robotis.com/docs/en/dxl/protocol2/#instruction-packet

#define INSTRUCTION\_PING 0x01 // checks if arriving packet ID is same as packet ID

#define INSTRUCTION\_READ 0x02 // read data from device

#define INSTRUCTION\_WRITE 0x03 // write data to device

#define INSTRUCTION\_REG\_WRITE 0x04 // registers instruction packet to set for standby

#define INSTRUCTION\_ACTION 0x05 // executes packet by INSTRUCTION\_ REG\_WRITE

#define INSTRUCTION\_FACTORY\_RESET 0x06 // reset Control Table to factory default

#define INSTRUCTION\_REBOOT 0x08 // reboot device

#define INSTRUCTION\_CLEAR 0x10 // reset certain information

#define INSTRUCTION\_STATUS\_RETURN 0x55 // return instruction for the Instruction packet

#define INSTRUCTION\_SYNC\_READ 0x82 // multiple devices: read all devices

#define INSTRUCTION\_SYNC\_WRITE 0x83 // multiple devices: write all devices

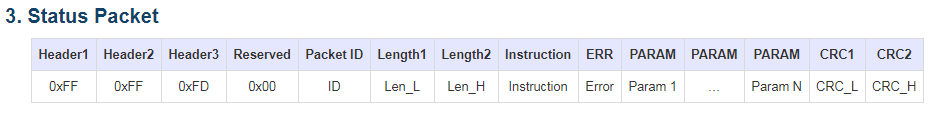
#define INSTRUCTION\_BULK\_READ 0x92 // multiple devices: read different devices

#define INSTRUCTION\_BULK\_WRITE 0x93 // multiple devices: write different devices

**Figure 1F**: #defines for the Robotis Dynamixel Protocol 2.0 Instruction Packet

**Step 4:** Create Definition Header File (H-File) – Packet Headers

Section 3 of Protocol 2.0 <http://emanual.robotis.com/docs/en/dxl/protocol2/#status-packet> provides details of packets. The XL-320’s firmware uses packets to read and/or write instructions. The packets are delivered via RS-485. A packet is information, typically in the form of bytes. Each byte and its location within the packet, connotes instructions. **Figure 1G** shows the packet form:



**Figure 1G:** [Section 3](http://emanual.robotis.com/docs/en/dxl/protocol2/#status-packet) of the Robotis Dynamixel Protocol 2.0 illustrates the packet format

The first 4 bytes do not change and hence included in the H-file. See (the un-highlighted section) of **Figure 1B**.

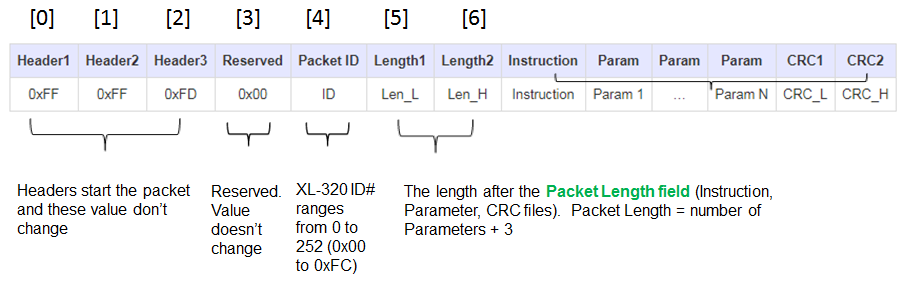
**Concept 2 Create Definition Header File (H-File) xl320-functions1\_0c.h**

**Preamble:**

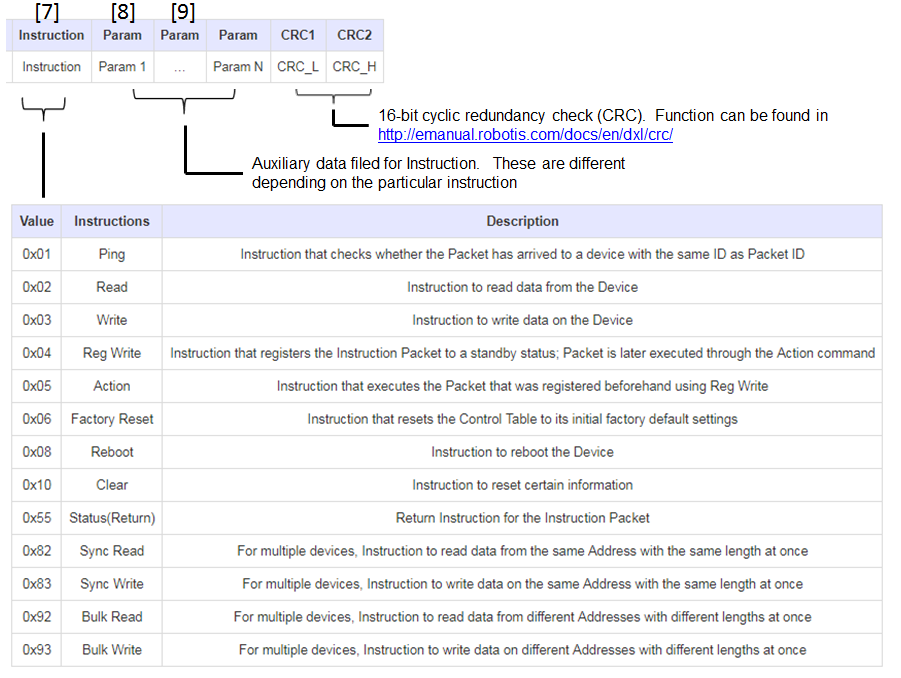
The XL-320’s firmware communicates via packets that have the form of **Figure 1G**. Code is simplified if one creates a separate H-file consisting of functions. The main C program can then call these functions. Function naming is not unique. Defining a naming convention would make code more readable; one could more easily recognize a function specifically written for the XL-320. For this Concept (and future ones), the convention used is the prefix XL320\_ and then a name (first word lower case and subsequent words’ first letter capitalized). For example, for lighting up the XL-320’s LED, the function would be:

void **XL320\_setLed**(unsigned char, unsigned char XL320\_ledColor)

**Packet Structure: Figure 2A** annotates **Figure 1G** with more details.



**Figure 2A:** Annotated explanation of each byte in a packet



**Figure 2A** continued – Table with Values is from **Figure 1E**

One notices that a packet length is not fixed. Some instructions require more information. As such, the Param field (see [9] in **Figure 2A**) could be multiple bytes. One references **Figure 1C** to determine how many bytes (and hence Parameters) will be needed.

Writing the **XL320\_setLed** function

**Step 1:** Calculate lengths Len\_L and Len\_H for the packet (see **Figure 2A** [5] and [6])

The XL-320’s LED

**Figure 1C** shows that the XL-320’s LED has an address of 25 Decimal (DEC). Also the size is 1-byte which can take a value from 0 to 7. Figure 2A shows that the packet length is the number of parameters + 3. One thus has:

Number of Parameters = Reserved Byte + Packet ID Byte + Value Byte + 3 Bytes = 6

The XL-320 uses 16-bit integer values. Thus to represent as two 8-bit numbers (i.e. 2 bytes), Protocol 2.0 employs a Little Endian format. Little Endian means that the lower significant bits are stored in the first byte, and the higher ones in the second byte. Hence to express 6 DEC (or 0x06) one uses sets Len\_L to 0x06 and Len\_H to 0x00. The yellow highlight in **Figure 2B** shows this expressed using the naming convention.

// Variables to set Length 1 and Length 2

unsigned char XL320\_setLedLength\_L;

unsigned char XL320\_setLedLength\_H;

// Variables to set up packet array

unsigned char tempPacket[]; // temporary packet

unsigned char finalPacket[]; // final packet to transmit

// Variables for checksum CRC

unsigned short setLed\_CRC;

byte CRC\_L, CRC\_H;

// 1. Calculate lengths

// Recall that Length 1 and Length 2 = number of parameters + 3

// Setting LED requires only 3 parameters: address, 0x00 and output color

// Hence number of parameters + 3 is 3 + 3 = 6 Dec = 0x06

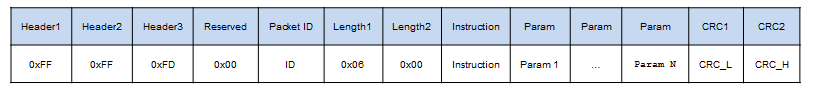
XL320\_setLedLength\_L = 0x06;

XL320\_setLedLength\_H = 0x00;

**Figure 2B:** Len\_L (or XL320\_setLedLength\_L) and Len\_H (or XL320\_setLedLength\_H) set to 0x06 and 0x00 respectively, represents a packet length of 6 DEC

**Step 2:** Construct first part of the Packet

Thus far, the packet for setting the LED looks like **Figure 2C**:



**Figure 2C:** From left to right: the first 4 bytes do not change.

The Packet ID is the servo ID number (which one sets under Dynamixel Wizard). The Length1 and Length2 bytes were calculated in the previous step.

In the definition H-file (**xl320-defines1\_0a.h**)one used #define for constants (see **Figure 2D**):

#define HEADER\_1 0xFF // For Instruction Packet Header 1

#define HEADER\_2 0xFF // For Instruction Packet Header 2

#define HEADER\_3 0xFD // For Instruction Packet Header 3

#define RESERVED 0x00 // For Instruction Packet Reserved

:

:

#define RAM\_LED 0x19 // 1 byte; changes motor's LED color

:

:

#define INSTRUCTION\_WRITE 0x03 // write data to device

**Figure 2D**: Snippets of xl320-defines1\_0a.h created in Concept 1

In NXC Programming, **Figure 2C** can be represented by the code snipped in **Figure 2E**. Here, the NXC function ArrayBuild is used to create a multi-byte packet called tempPacket.

// 2. Construct first part of packet

ArrayBuild(tempPacket, HEADER\_1, HEADER\_2, HEADER\_3, RESERVED, XL320\_servoId,

XL320\_setLedLength\_L, XL320\_setLedLength\_H, INSTRUCTION\_WRITE,

RAM\_LED, 0x00, XL320\_ledColor);

**Figure 2E:** Code snippet of **xl320-functions1\_0c.h**

In yellow-highlight are variables that are passed to the XL320\_setLed function, namely, the servo’s ID number and desired color.

**Step 3:** Perform Checksum

Checksum is often employed to verify packets were correctly transferred between devices. The XL-320 uses the Cyclic Redundancy Check (CRC) method to perform the checksum as seen in <http://emanual.robotis.com/docs/en/dxl/crc/>. The Robotis XL-320 E-Manual provides this function in C as shown in **Figure 2F**.



**Figure 2F:** [Section 1](http://emanual.robotis.com/docs/en/dxl/crc/) of the CRC Calculation from Robotis E-Manual

The details of how CRC operates are beyond the scope of this lab. Essentially, an NXC equivalent of **Figure 2F** is created as a function update\_crc in the xl320-functions1\_0c.nxc file.

The function is called with the following NXC code:

// 3. Perform checksum, see Section 1.2 of http://emanual.robotis.com/docs/en/dxl/crc/

unsigned int packetLength = (XL320\_setLedLength\_H >> 8) + XL320\_setLedLength\_L;

// See last bullet in Section 1.2 "Packet Analysis and CRC Calculation"

setLed\_CRC = update\_crc(0, tempPacket, 5 + packetLength);

CRC\_L = (setLed\_CRC & 0x00FF);

CRC\_H = (setLed\_CRC >> 8) & 0x00FF;

**Figure 2G:** CRC values are called using the update\_crc function call

Note that the logic AND operator and bit-wise operator >> are used to create the Little Endian forms of the low and high bytes for CRC\_L and CRC\_H respectively.

**Step 4:** Concatenate final packet and transmit

Now that the CRC bytes have been calculated, the final form of the packet can be built, transmitted via a NXC RS485Write call and confirmed via a NXC waitForMessageToBeSent call (see **Figure 2H**).

// 4. Concatenate into final packet and sent thru NXT RS485

ArrayBuild(finalPacket, tempPacket, CRC\_L, CRC\_H);

RS485Write(finalPacket);

// 5. Call inline function

waitForMessageToBeSent();

}; // end XL320\_setLed function

**Figure 2H:** Final packet creation, transmission and confirmation

**Concept 3 Create NXC Main program xl320-helloLed1\_0a.nxc**

With firmware constants defined (Concept 1) and the packet formation for the LED (Concept 2) H-files ready, one can write a simple NXC program to light the XL-320’s LED.

**Step 1:** Start NXC code with comments and #includes to H-files

// FILE: xl320-helloLed1\_0a.nxc - Works!

// DATE: 09/19/19 13:06

// AUTH: P.Oh

// DESC: Cycles thru XL-320 LED colors

// VERS: 1.0a:

// REFS: xl320-setLed1\_0b.nxc; xl320-functions1\_0a.h; xl320-defines.h

// NOTE: If factory default XL-320 used, then ID is 0x01

// ID of 0xFE commands any and all XL-320 motors

#include "xl320-defines1\_0a.h" // XL-320 defines from Control Table

#include "xl320-functions1\_0c.h" // P.Oh functions written for XL-320

#define ID\_ALL\_MOTORS 0XFE // 0XFE commands all XL-320 motors

#define ID\_MOTOR01 0X01 // Assumes Motor 1 configured with ID = 1

The yellow-highlights shows include the H-files from Concepts 1 and 2. The ID\_MOTOR01 is defined as a constant. This should be the Servo ID number that was set in Dynamixel Wizard (e.g. 0x01). A servo ID number of 0xFE is defined by the XL-320 firmware as commanding any and all connected servos.

**Step 2:** Declare NXT buttons and establish RS-485 connections

task main() {

byte ledColor;

bool orangeButtonPushed; // Detect Brick Center button state

bool rightArrowButtonPushed; // Detect Brick right arrow button state

bool leftArrowButtonPushed; // Detect Brick left arrow button state

UseRS485();

RS485Enable();

// Note: First, use Dynamixel Wizard to set XL-320 to desired baud rate

// Then, use RS485Uart to match this baud rate e.g. 9600

RS485Uart(HS\_BAUD\_57600, HS\_MODE\_8N1); // 57600 baud, 8bit, 1stop, no parity

ClearScreen();

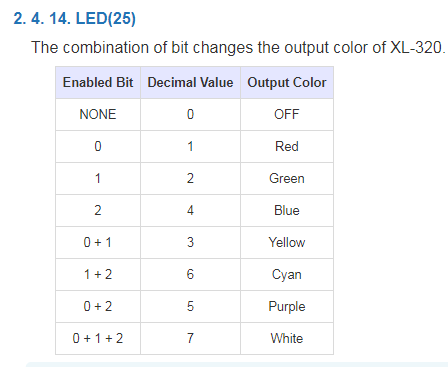
// Prompt user to begin

TextOut(0, LCD\_LINE1, "Stop: Press ORG" );

The program will detect pushing of the NXT Brick’s center button (orange-colored one) to commence cycling thru LED colors. The NXC RS-485 functions UseRS485, RS485Enable, and RS485Uart establish a 57600 baud rate and 8N1 (8-bits, no-parity, 1 stop-bit) protocol.

**Step 3:** Call XL320\_setLed function to change LED color

Section 2.4.14 of the XL-320 E-manual <http://emanual.robotis.com/docs/en/dxl/x/xl320/#led> details the different LED colors on the servo (see **Figure 3A**)



**Figure 3A:** [Section 2.4.14](http://emanual.robotis.com/docs/en/dxl/x/xl320/#led) of the XL-320 E-Manual shows LED color values

There are 8-states for the XL-320 LED: off or 7 different colors. A switch statement, in a do-while loop, cycles thru the colors.

// See Section 2.4.14 LED http://emanual.robotis.com/docs/en/dxl/x/xl320/#led

// LED values: 0 (Off); 1 (Red); 2 (Green); 3 (Yellow); 4 (Blue); 5 (Purple)

// 6 (Cyan); 7 (White)

ledColor = 0; // set LED to off first

do {

orangeButtonPushed = ButtonPressed(BTNCENTER, FALSE);

XL320\_setLed(ID\_ALL\_MOTORS, ledColor);

switch(ledColor) {

case 0: TextOut(0, LCD\_LINE3, FormatNum("%d OFF" , ledColor));

break;

case 1: TextOut(0, LCD\_LINE3, FormatNum("%d RED" , ledColor));

break;

case 2: TextOut(0, LCD\_LINE3, FormatNum("%d GRN" , ledColor));

break;

case 3: TextOut(0, LCD\_LINE3, FormatNum("%d YLW" , ledColor));

break;

case 4: TextOut(0, LCD\_LINE3, FormatNum("%d BLU" , ledColor));

break;

case 5: TextOut(0, LCD\_LINE3, FormatNum("%d PUR" , ledColor));

break;

case 6: TextOut(0, LCD\_LINE3, FormatNum("%d CYA" , ledColor));

break;

case 7: TextOut(0, LCD\_LINE3, FormatNum("%d WHT" , ledColor));

break;

}; // end switch

Wait(1000);

ledColor++;

if(ledColor > 7) ledColor = 0;

} while(!orangeButtonPushed);

ClearScreen();

} // end main

The yellow-highlighted line shows the call to XL320\_setLed, which passes the servo’s ID number and desired color). The color name is displayed based on values in **Figure 3A**.

Congratulations! You can change the XL-320 LED color.