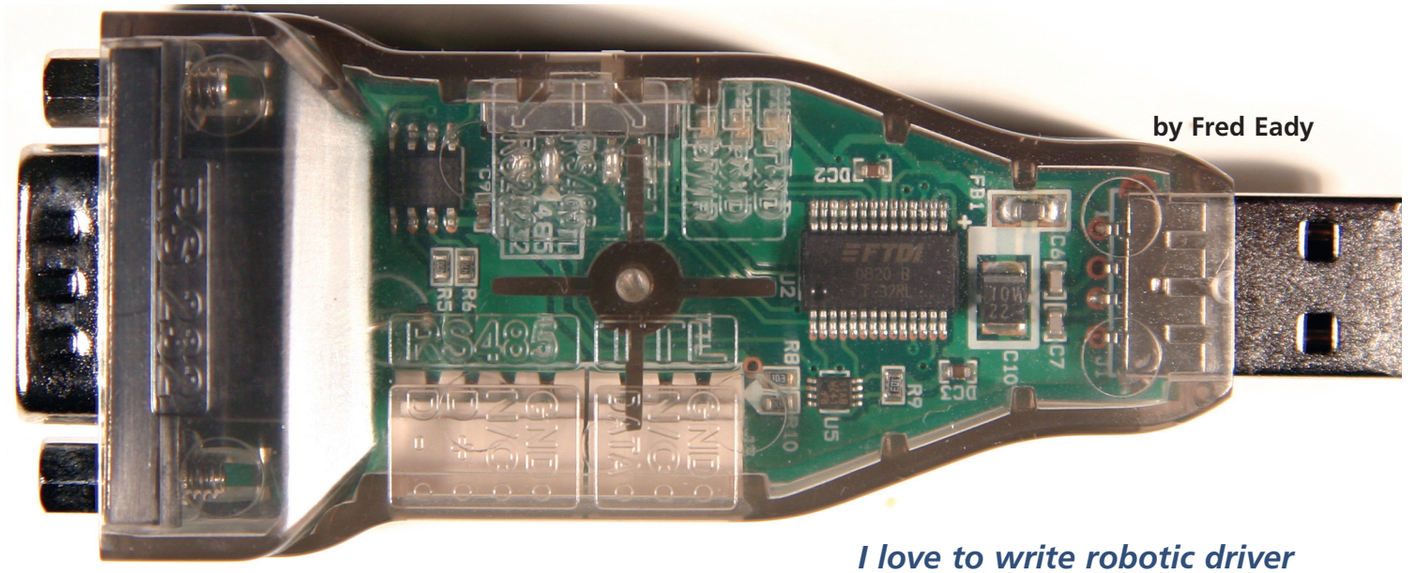


by Fred Eady

UNWINDING THE AX-12+ COMMUNICATION PROTOCOL

I love to write robotic driver firmware and scratch build PIC microcontroller-based robotic hardware to run it. In this edition of SERVO, we're not only going to sharpen our driver authoring skills, we'll also get some flight time on the handle of a soldering iron.

In the discussion and hardware build-up that follows, our collective programming and hardware design/assembly efforts will be focused on the Microchip PIC18F2620, which will be coded to drive a Dynamixel AX-12+ robot actuator. There are several Dynamixel robot actuators in addition to the AX-12+. This month, we will center exclusively on instructing the F2620 to drive a Dynamixel AX-12+ robot actuator. We've got some specialized AX-12+ hardware to design and assemble before we can begin to code the driver firmware. So, let's get started.

The Dynamixel AX-12+

I can easily describe the Dynamixel AX-12+ robot actuator you see in Photo 1 with a single word: SuperServo. The AX-12+ can do everything a standard hobby servo can and better. For instance, to obtain continuous rotation you don't have to disassemble and intentionally "break" it. You simply command it to perform an endless turn. Need to know where the servo shaft is? Don't ask a hobby servo, because unless it's one of the new digital models, it can't tell you. A Dynamixel can not only tell you where its shaft is, it can also tell you if it's moving.

In that you're reading this magazine, odds are that you have some prior exposure to hobby servos. For those experienced readers, you know that hobby servos move at their own mechanical pace with a set amount of torque. The Dynamixel can be

PHOTO 1. The daisy-chained one-wire RS-232 link I am referring to is actually a DATA line and a GROUND line. The AX-12+ also distributes power on a third daisy-chained line.

commanded to move at *your* selected angular velocity with *your* desired amount of torque to within ±0.35° of the desired endpoint position. The AX-12+'s maximum rated holding torque is 229 ounce-inches and it can rotate at a maximum angular velocity of 114 rpm. Needless to say, if you get one of your humanoid body parts in the way of a high-speed max-torque AX-12+ mechanical operation, it's gonna leave a mark.

Standard hobby servos use a variable duty cycle pulse train to control their shaft's angular velocity and position. The duty cycle of the servo control pulse determines the servo shaft's rotational position while the angular velocity of the servo shaft is dictated by the speed of the duty cycle modulation. Thus, the slower the duty cycle change, the slower the angular velocity. A pulse width of 1.0 ms will move a standard hobby servo shaft to the extreme left, while a 2.0 ms pulse will move the shaft to the far right. Centering the shaft requires a pulse with a width of 1.5 ms.

When a flock of hobby servos need to be individually positioned to achieve a common goal such as in model aircraft and boats, each servo must have access to its unique pulse width information. In these cases, the unique pulse widths are multiplexed by a transmitter and demultiplexed at the receiver. If the hobby servos aren't in the air or on the water, an elaborate microcontroller-based multiple pulse width generator is normally used to control the servo positions.

The Dynamixel robot actuators don't depend on pulse widths for their position information. Instead, a half-duplex, one-wire, RS-232 protocol-based TTL communications link transfers command and status information between a host controller and the robot actuators. The TTL-level status and command messages are called digital packets. The term half-duplex means that devices attached to a common communications link are only allowed to talk when no other device is talking. In the case of the Dynamixel robot actuators, all of the robot actuators that are daisy-chained on the one-wire link spend most of their time listening and only speak after being spoken to.

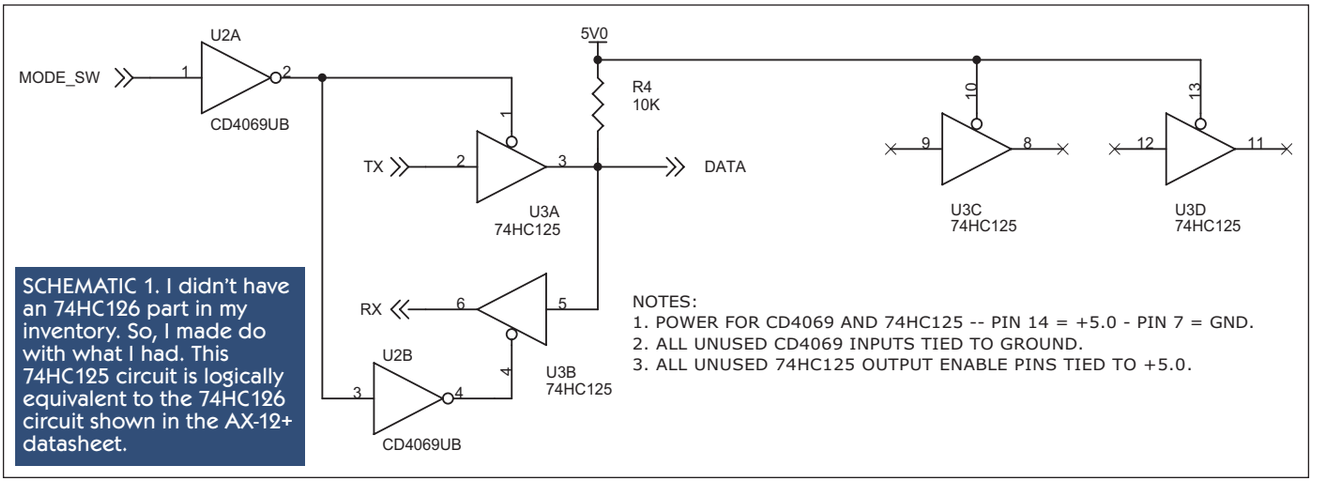
It is also possible to command the robot actuators in the daisy chain to listen and obey only. All of the Dynamixel robot actuators on the link are able to hear every message that is transmitted on the wire. However, each actuator that participates on the half-duplex one-wire TTL link is assigned a unique address between 0 and 253 decimal. If an actuator hears a message that does not contain its assigned address, the message is ignored. The only way to get the attention of every AX-12+ on the link at the same time is to send a digital packet using the broadcast address, which is 254 decimal (0xFE).

In addition to carrying precision position and speed information, the digital packets can also transport robot actuator feedback data. We already know that with the issuance of a command from the host controller, an AX-12+ can report its angular position and/or its angular velocity. Other robot actuator parameters such as internal temperature, input voltage, and load torque can also be queried by the host controller. The fact of the matter is, we can issue a single READ command and gain access to all of the data held in the AX-12+'s Control Table.

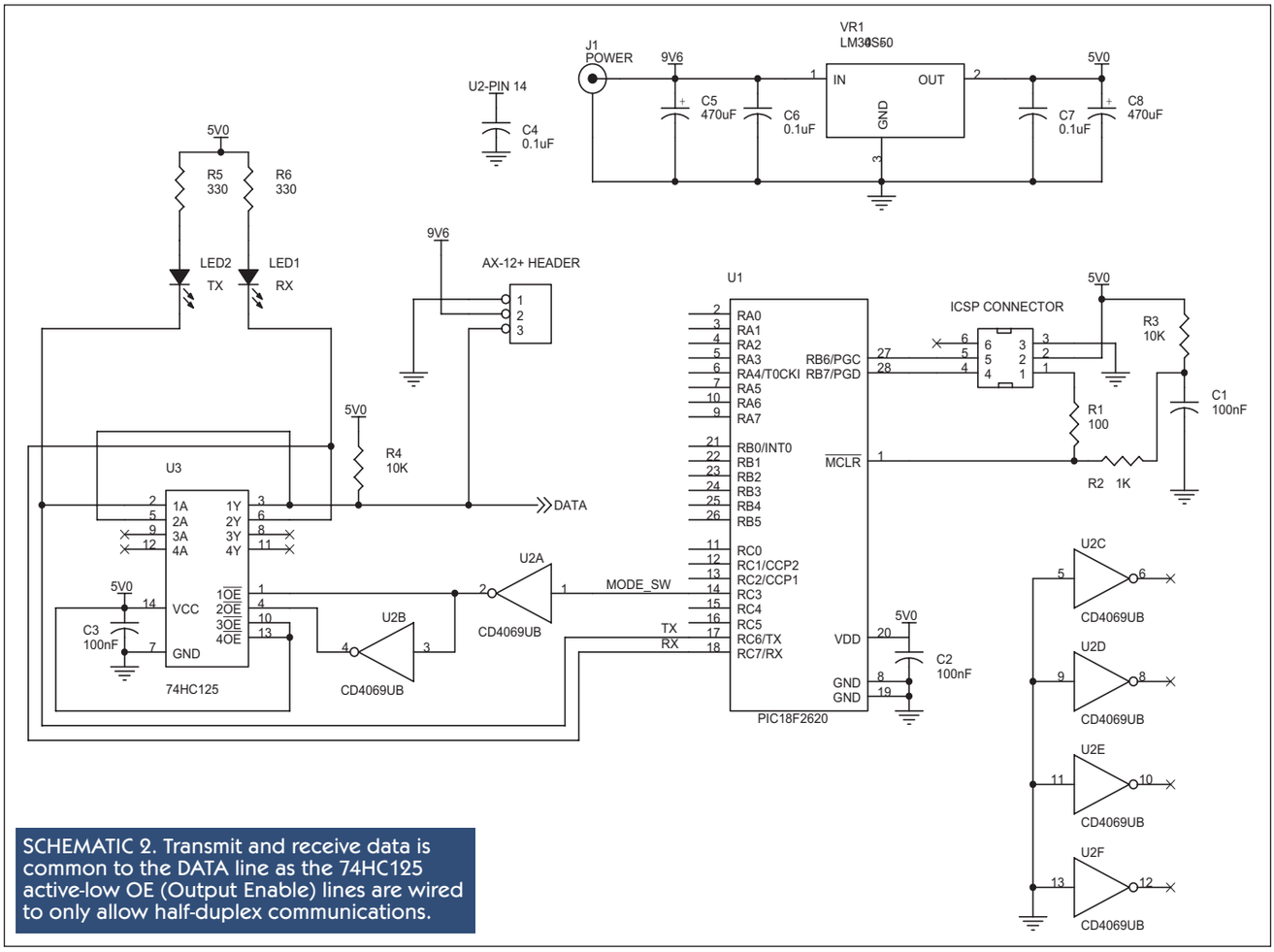
I could expound on the virtues of the AX-12+ all day. However, you're not here to listen to me talk. You're here to get the skinny on how to put an AX-12+ to work under the control of a PIC18F2620. With that, let's determine what we need in a hardware way to get the PIC and an AX-12+ to communicate with each other.

An AX-12+ Controller Hardware Design

Behold Schematic 1. I've used a pair of CD4069 inverter gates to mirror the half-duplex transmit/receive logic that is set forth by the AX-12+ datasheet. A TTL high applied to the *MODE_SW* inverter input enables U3A, the transmit buffer, and tristates the output of U3B, the receive buffer. Conversely, a TTL low at the *MODE_SW* input tristates the transmit buffer's output and enables the receive buffer output of U3B. This simple circuit is the key to the implementation of the one-wire half-duplex TTL link required by the AX-12+. The AX-12+ datasheet presents the



SCHEMATIC 1. I didn't have an 74HC126 part in my inventory. So, I made do with what I had. This 74HC125 circuit is logically equivalent to the 74HC126 circuit shown in the AX-12+ datasheet.



SCHEMATIC 2. Transmit and receive data is common to the DATA line as the 74HC125 active-low OE (Output Enable) lines are wired to only allow half-duplex communications.

circuit you see in Schematic 1 using a 74HC126. I didn't have a 74HC126 in my IC inventory. Being that the only difference between the 74HC125 and the 74HC126 is the active polarity of the tristate control inputs, I added the inverter U2A to make the 74HC125 appear as a 74HC126 to the firmware logic. Since we're writing our own AX-12+ driver, we could dispense with U2A and run the transmit/receive switching logic in the reverse direction of the AX-12+ datasheet. However, to maintain continuity with the AX-12+ system logic, it's best to keep with the datasheet logic and perform the switching logic inversion with U2A.

Schematic 2 adds the detail you need to envision the entire one-wire interface and its interconnection with the PIC18F2620's I/O subsystem. The communications link *DATA* portal at pin 3 of the 74HC125 connects to the AX-12+'s physical DATA pin, whose logical state can be transmitted via daisy chain to other AX-12+ DATA pins on the link. Note that the +9.6 volt bulk motor voltage and the common ground are also included in the daisy chain link.

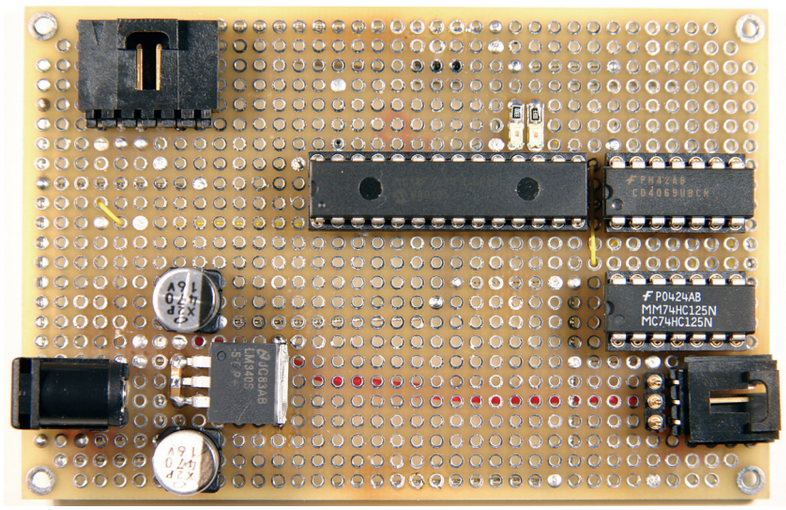
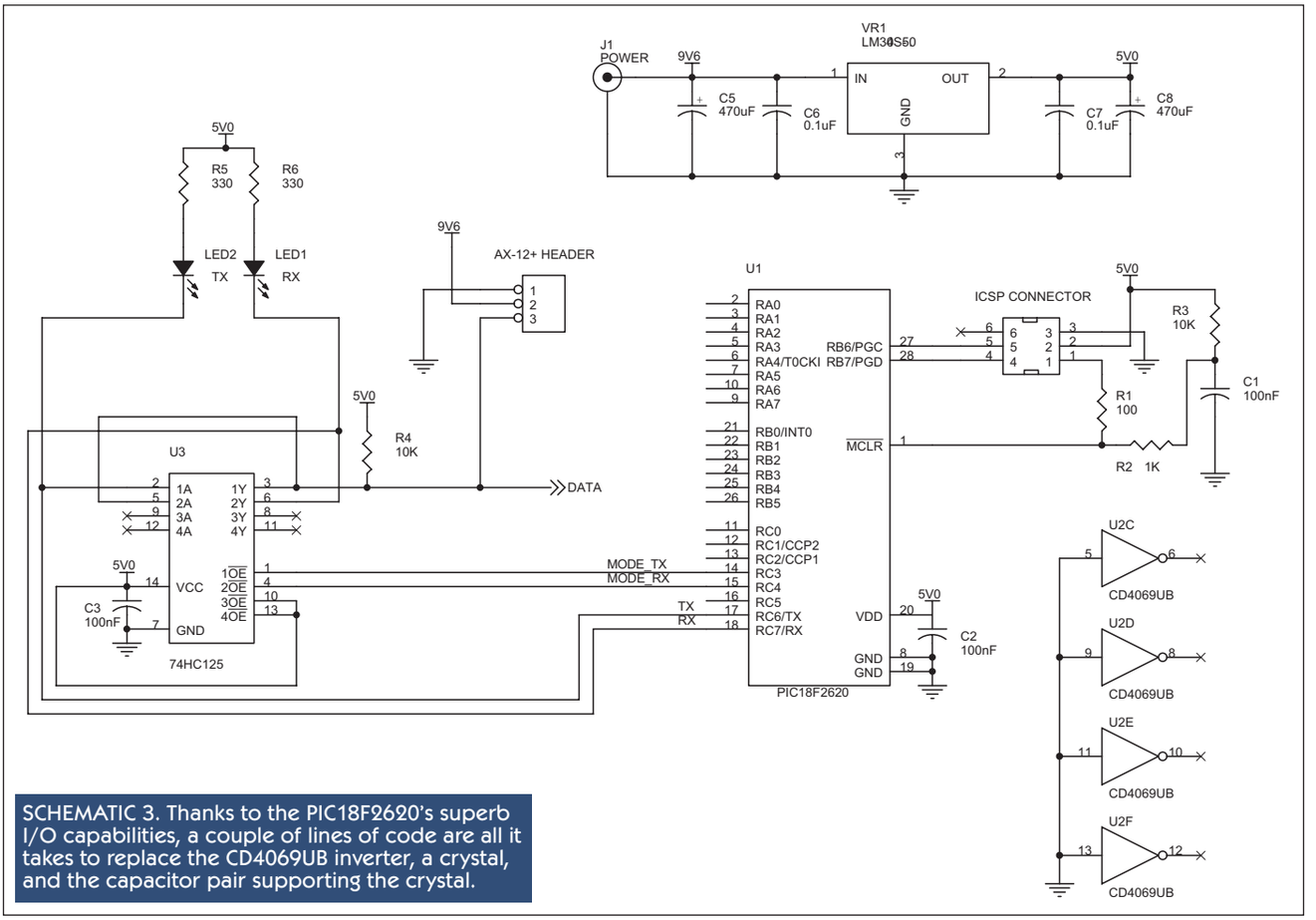


PHOTO 2. The circuitry for the SuperServo is SuperSimple. So, a printed circuit board is not necessary. I used standard point-to-point solder techniques to wire up this AX-12+ controller design. The three-wire interface for the AX-12+ is made up of a portion of a SIP header strip.



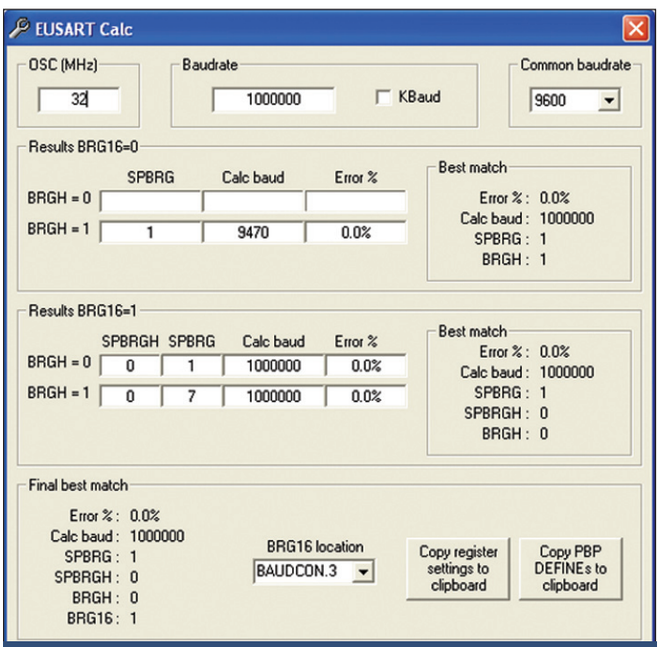
SCHEMATIC 3. Thanks to the PIC18F2620's superb I/O capabilities, a couple of lines of code are all it takes to replace the CD4069UB inverter, a crystal, and the capacitor pair supporting the crystal.

Each AX-12+ attached to the common link can draw up to 900 mA. So, we need to make sure we provide a +9.6 volt power source that is hefty enough to support every AX-12+ in the daisy chain.

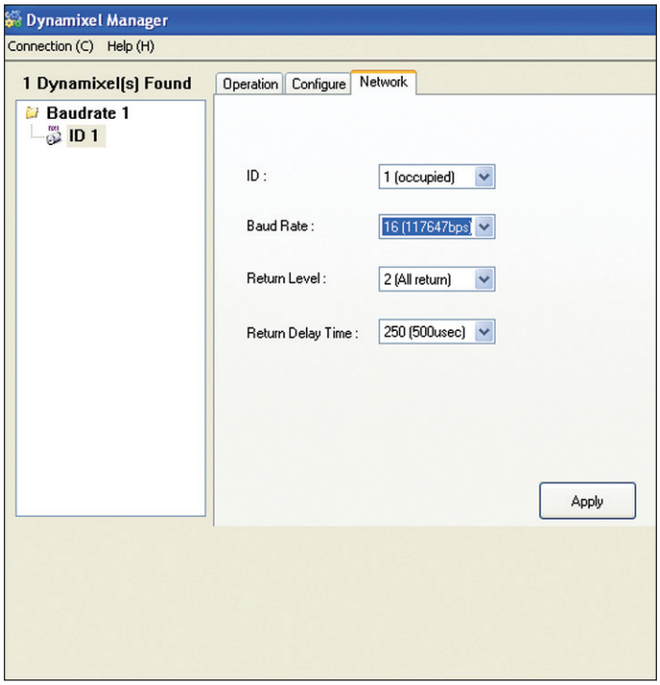
If you're wondering what happened to the PIC18F2620's crystal, it is not necessary in this design as we will be coding in the internal 32 MHz clock. We can conserve I/O pins by building up the design you see in Schematic 2. If I/O will be plentiful in your design and you want to eliminate the CD4069, you can. Take a look at Schematic 3. We have simply given direct control of the 74HC125 OE (Output Enable) pins to the PIC18F2620. The only caveat in this design is that you must make sure that you switch the PIC18F2620's *MODE_TX* and *MODE_RX* I/O lines correctly in the firmware. As you can see in Photo 2, I've gone with the hardware-heavy Schematic 2 design. If you decide to go with the Schematic 3 design, we'll code in and comment out the necessary mode switch code in the AX-12+ driver firmware.

The TX and RX LEDs take advantage of the PIC18F2620 EUSART's logically high idle state. The LEDs will blink with every passing logic low on the communications link. Thus, you'll see every START bit and every binary zero in the data stream in the lights.

There's no rocket science in the power supply or the



SCREENSHOT 1. I recommend adding this little utility to your programming arsenal. Although it looks like the original site is gone, search the web using PicMultiCalc and you'll find archives that will allow you to download the executable.



SCREENSHOT 2. Exploring the functionality of this software tool with a USB2Dynamixel and AX-12+ attached to your PC's USB port is a quick and easy way to get acquainted with the Dynamixel robot actuator system.

ICSP portal. Strip away the 74HC125 and CD4069 circuitry and this becomes a baseline PIC design.

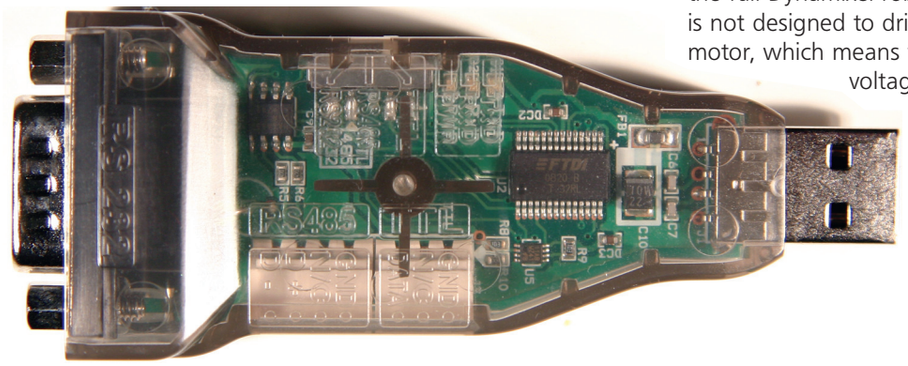
Driving the AX-12+ with a PIC18F2620

The AX-12+ comes from the factory addressed as 0x01 and ready to communicate at its maximum speed of 1 Mbps. If you don't believe the PIC18F2620's EUSART can run with the big dogs at 1 Mbps, swing your eyes over to Screenshot 1. The math is courtesy of PicMultiCalc. This PIC utility runs on a PC and is the brainchild of Mister E Engineering. The absence of the PicMultiCalc download on the Mister E website seems to indicate that Mister E is out of the country at the moment. However, I did find a downloadable copy of the utility in the microEngineering Labs PICBASIC forum. Here's what our EUSART initialization code looks like after incorporating the PicMultiCalc numbers:

```
//*******************************************************
//*      Init EUSART Function
//*******************************************************
void init_EUSART(void)
{
  SPBRG =  7;   //7 = 1 Mbps with 32MHZ clock
  //SPBRG =  68; //68 = 115200 bps with 32MHZ clock
  TRISC7 = 1;   //receive pin
  TRISC6 = 0;   //transmit pin
  BRG16 = 1;
  TXSTA = 0x04; //high speed baud rate set BRGH = 1
  RCSTA = 0x80; //enable serial port and pins
  EUSART_RxTail = 0x00; //flush Rx buffer: Head=Tail
  EUSART_RxHead = 0x00;
  EUSART_TxTail = 0x00; //flush Tx buffer: Head = Tail
  EUSART_TxHead = 0x00;
  RCIP = 1;     //receive interrupt = high priority
  TXIP = 1;     //transmit interrupt = high priority
  RCIE = 1;     //enable receive interrupt
  PEIE = 1;     //enable all unmasked peripheral irqs
  GIE = 1;      //enable all unmasked interrupts
  CREN = 1;     //enable EUSART1 receiver
  TXIE = 0;     //disable EUSART1 transmit interrupt
  TXEN = 1;     //transmitter enabled
}
```

Note that I added a commented-out *SPBRG* statement for 115200 bps. The reasoning behind this is that I initially tested the AX-12+ driver at 115200 thinking that 1 Mbps was not a realistic baud rate for the PIC18F2620. If you want to experiment with other baud rates, you'll need to preload the out-of-the-box AX-12+ with your desired baud rate. The easiest way to do this is to use a Robotis USB2Dynamixel dongle like the one smiling at you in Photo 3.

PHOTO 3. If you've been keeping up with my RS-232 to USB conversion projects in *Nuts & Volts*, you already know quite a bit about what's going on inside of the USB2Dynamixel. The USB2Dynamixel is designed around the FTDI FT232R USB UART IC.



The USB2Dynamixel is based on the FTDI FT232R USB UART IC. With the flick of a slide switch, the USB2Dynamixel can transform a PC's USB data stream into RS-232, RS-485, or TTL voltage levels suitable for use with the full Dynamixel robot actuator line. The USB2Dynamixel is not designed to drive the robot actuator electronics and motor, which means that we have to supply the bulk motor voltage if we wish to exercise the AX-12+ using the USB2Dynamixel. The USB2Dynamixel is supported by a number of PC-based control and test programs, which can be downloaded from the Robotis website.

The AX-12+-USB2Dynamixel hardware hookup was simple. I crimped a pair of Molex female terminals (Molex part number

16-02-1125) onto the opposite end of a couple of wires that I ultimately connected to a +9.6 volt power source. The AX-12+ comes with a three-wire jumper that I used to connect it to the USB2Dynamixel. I plugged the USB2Dynamixel into my laptop serial port, powered up the +9.6 volt supply, and kicked off the Dynamixel Manager application.
As you can see in Screenshot 2 — with the help of the USB2Dynamixel and a home-made power cable — I used the Network portion of the Dynamixel Manager to preload the 115200 baud rate into the AX-12+. Naturally, I used the same process to return the AX-12+ to its original 1 Mbps baud setting. We don't need to run the PIC18F2620's internal oscillator at full blast to pump 1 Mbps out of its EUSART. According to PicMultiCalc, we can run with an 8 MHz clock and still achieve 1 Mbps throughput at the EUSART. Let's settle on running at full blast, which is 32 MHz. The clock coding for 32 MHz goes like this:

```
//*******************************************************
//*     INITIALIZE CLOCK AND I/O PORTS
//*******************************************************
        OSCCON = 0x70;  //set for 32 MHz operation
        PLLEN = 1;      //enable PLL for 32 MHz
        TRISA = 0b01111111;
        TRISB = 0b11111111;
        TRISC = 0b10000000;
```

All of the I/O that relates to the AX-12+ is currently taking place on PORT C of the PIC18F2620. To get things moving as quickly as possible, I like to set the I/O port directions as soon as I can in the code.

To send a digital packet, we must know how the data is laid out within each packet. So, let's look at a digital packet from the viewpoint of a programmer using the C programming language. We'll stash our digital packets into a 128-byte array called *xmit_buff* until we're ready to send them.

Every digital packet begins with a pair of 0xFF sync characters:

```
xmit_buff[0] = 0xFF;     //sync character
xmit_buff[1] = 0xFF;     //sync character
```

Since our EUSART firmware engine uses circular buffers to hold its transmit and receive data, the pair of 0xFF sync characters will always stand as digital packet demarcation points. The byte that immediately follows the sync characters is the AX-12+ ID, which ranges from 0 to 253 decimal (0x00 to 0xFE). There will never be a trio of consecutive 0xFF characters as the ID can never be greater than 0xFE. So, we're still safe triggering on a pair of 0xFF characters to denote the beginning bytes of a digital packet. Here's what our digital packet looks like so far:

```
xmit_buff[0] = 0xFF;     //sync character
xmit_buff[1] = 0xFF;     //sync character
xmit_buff[2] = id;       //unique ID 0-253
```

The next byte in a digital packet holds a number representing the length of the digital packet, which is computed as the Number of Parameters + 2:

```
xmit_buff[0] = 0xFF;             //sync character
xmit_buff[1] = 0xFF;             //sync character
xmit_buff[2] = id;               //unique ID
xmit_buff[3] = parm_len + 2;     //PARMS+INSTR+CHECKSUM
```

The additional two bytes added to the parameter length include the instruction and the digital packet checksum value in the packet length calculation. The parameters — if there are any — are squeezed in between the instruction and checksum bytes:

```
xmit_buff[0] = 0xFF;             //sync character
xmit_buff[1] = 0xFF;             //sync character
xmit_buff[2] = id;               //unique ID
xmit_buff[3] = parm_len + 2;     //PARMS-INSTR-CHECKSUM
xmit_buff[4] = inst;             //instruction
xmit_buff[p] = parms[0],parms[1] //any number of params
xmit_buff[c] = packet checksum   //packet checksum byte
```

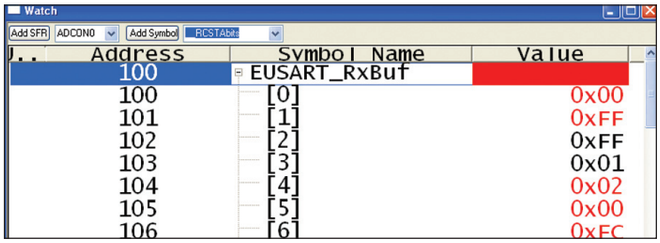
Here's how we define the seven AX-12+ instructions in our firmware:

```
//*******************************************************
//*     INSTRUCTIONS
//*******************************************************
#define iPING           0x01 //obtain a status packet
#define iREAD_DATA      0x02 //read Control Table values
#define iWRITE_DATA     0x03 //write Control Table values
#define iREG_WRITE      0x04 //write and wait for ACTION
#define iACTION         0x05 //triggers REG_WRITE
#define iRESET          0x06 //set factory defaults
#define iSYNC_WRITE     0x83 //control mult. actuators
```

The instruction parameters are kept in their own 128-byte array, which we call *parms*. Let's dry-run some example digital packets to show you how the *parms* array works with the *xmit_buff* array. We'll begin with building a PING digital packet, which has no parameters. The AX-12+ ID will be 0x01 in all of our examples:

```
//PING DIGITAL PACKET
xmit_buff[0] = 0xFF;             //sync character
xmit_buff[1] = 0xFF;             //sync character
xmit_buff[2] = 0x01;             //unique ID
xmit_buff[3] = 0x02;             //number of PARMS+
                                 //INSTRUCTION+CHECKSUM
xmit_buff[4] = iPING;            //instruction
xmit_buff[5] = 0xFB;             //checksum
```

The only byte you probably can't figure out right now is held in xmit_buff[5]. The digital packet checksum is simply the bitwise inversion (logical NOT) of the sum of the ID byte, length byte, parameter bytes, and instruction byte.



SCREENSHOT 3. This is a capture of the PIC18F2620's EUSART receive buffer. A value of 0x00 in the ERROR byte is a very good thing as bits that are set indicate errors.

Any bits that roll out of the checksum's least significant byte are ignored.

Let's dry-run with an instruction that requires some parameters. Let's manually code up a READ_DATA digital packet that will retrieve the AX-12+'s ID from the Control Table. The Control Table is simply a chunk of EEPROM and RAM that holds the AX-12+'s configuration and feedback data. The first 23 Control Table entries are nonvolatile. There are 49 Control Table memory slots:

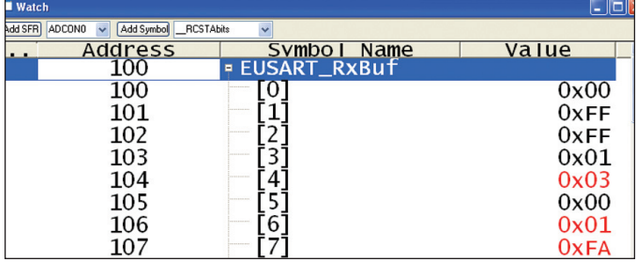
```
//*****************************************************
//*      CONTROL TABLE ADDRESSES
//*****************************************************
enum{
  MODEL_NUMBER_L,              // 0x00
  MODEL_NUMBER_H,              // 0x01
  VERSION,                     // 0x02
  ID,                          // 0x03
  BAUD_RATE,                   // 0x04
  RETURN_DELAY_TIME,           // 0x05
  CW_ANGLE_LIMIT_L,            // 0x06
  CW_ANGLE_LIMIT_H,            // 0x07
  CCW_ANGLE_LIMIT_L,           // 0x08
  CCW_ANGLE_LIMIT_H,           // 0x09
  RESERVED1,                   // 0x0A
  LIMIT_TEMPERATURE,           // 0x0B
  DOWN_LIMIT_VOLTAGE,          // 0x0C
  UP_LIMIT_VOLTAGE,            // 0x0D
  MAX_TORQUE_L,                // 0x0E
  MAX_TORQUE_H,                // 0x0F
  STATUS_RETURN_LEVEL,         // 0x10
  ALARM_LED,                   // 0x11
  ALARM_SHUTDOWN,              // 0x12
  RESERVED2,                   // 0x13
  DOWN_CALIBRATION_L,          // 0x14
  DOWN_CALIBRATION_H,          // 0x15
  UP_CALIBRATION_L,            // 0x16
  UP_CALIBRATION_H,            // 0x17
  TORQUE_ENABLE,               // 0x18
  LED,                         // 0x19
  CW_COMPLIANCE_MARGIN,        // 0x1A
  CCW_COMPLIANCE_MARGIN,       // 0x1B
  CW_COMPLIANCE_SLOPE,         // 0x1C
  CCW_COMPLIANCE_SLOPE,        // 0x1D
  GOAL_POSITION_L,             // 0x1E
  GOAL_POSITION_H,             // 0x1F
  MOVING_SPEED_L,              // 0x20
  MOVING_SPEED_H,              // 0x21
  TORQUE_LIMIT_L,              // 0x22
  TORQUE_LIMIT_H,              // 0x23
  PRESENT_POSITION_L,          // 0x24
  PRESENT_POSITION_H,          // 0x25
  PRESENT_SPEED_L,             // 0x26
  PRESENT_SPEED_H,             // 0x27
  PRESENT_LOAD_L,              // 0x28
  PRESENT_LOAD_H,              // 0x29
  PRESENT_VOLTAGE,             // 0x2A
  PRESENT_TEMPERATURE,         // 0x2B
  REGISTERED_INSTRUCTION,      // 0x2C
  RESERVE3,                    // 0x2D
  MOVING,                      // 0x2E
  LOCK,                        // 0x2F
  PUNCH_L,                     // 0x30
  PUNCH_H                      // 0x31
};
```

To access the AX-12+ ID byte, we need to build a READ_DATA digital packet to retrieve the fourth Control Table byte. Since the READ_DATA instruction requires parameters, the first step involves defining the parameter table in the *parms* array:

```
parms[0] = ID;   //starting read address
parms[1] = 0x01; //number of bytes to read from
                 // starting read address
```

Later on, we will write a function to load the *parm*

SCREENSHOT 4. The payload data (AX-12+ ID) is loaded at offset 6 in the EUSART's receive buffer.



Sources

- ***ROBOTIS — www.robotis.com***
  USB2Dynamixel; AX-12+ Robot Actuator; Dynamixel Manager
- ***Microchip — www.microchip.com***
  PIC18F2620; MPLAB IDE; MPLAB REAL ICE
- ***HI-TECH Software — www.htsoft.com***
  HI-TECH PICC-18 C Compiler

The AX-12+ firmware driver was compiled with HI-TECH PICC-18 PRO.

A Microchip MPLAB REAL ICE was used as the debugging device.

array entries into the correct memory slots of a digital packet. However, for now, let's insert the parameters manually:

```
//READ ID DIGITAL PACKET
xmit_buff[0] = 0xFF;     //sync character
xmit_buff[1] = 0xFF;     //sync character
xmit_buff[2] = 0x01;     //unique ID
xmit_buff[3] = 0x04;     //# of PARMS+INSTRUCTION+CHECKSUM
xmit_buff[4] = iREAD_DATA;       //instruction
xmit_buff[5] = ID;       //parameter 1
xmit_buff[6] = 0x01;     //parameter 2
xmit_buff[7] = 0xF4;     //checksum
```

Both of the digital packets we assembled will trigger a response from the AX-12+. In the case of the PING digital packet, we should receive a status message containing the AX-12+'s ID, an ERROR byte, and a checksum byte. If the PING operation is successful, here's what a returned status digital packet looks like from a programmer's point of view:

```
xmit_buff[0] = 0xFF;     //sync character
xmit_buff[1] = 0xFF;     //sync character
xmit_buff[2] = 0x01;     //payload value = ID of AX-12+
xmit_buff[3] = 0x02;     //# of PARMS+INSTRUCTION+CHECKSUM
xmit_buff[4] = 0x00;     //error byte - 0x00 = none
xmit_buff[5] = 0xFC;     //checksum
```

This status message contents are confirmed in Screenshot 3, which is the PING response data I received from an AX-12+ with an ID of 0x01. I also took the liberty to capture the response for the READ_DATA digital packet in Screenshot 4. Here's the programmer view of the status message data captured in Screenshot 4:

```
xmit_buff[0] = 0xFF;     //sync character
xmit_buff[1] = 0xFF;     //sync character
xmit_buff[2] = 0x01;     //unique ID
xmit_buff[3] = 0x03;     //# of PARMS+INSTRUCTION+CHECKSUM
xmit_buff[4] = 0x00;     //ERROR byte - no errors
xmit_buff[5] = 0x01;     //parameter 1
xmit_buff[6] = 0xFA;     //checksum
```

More To Come

I think you've got the idea. So, next time we'll add some meat to our firmware potatoes and code up a number of functions that will give us dominion over the AX-12+ Dynamixel robot actuator. In the meantime, I'll post the preliminary AX-12+ PIC18F2620 firmware I used here to communicate with an AX-12+ as a download package on the *SERVO* website (**www.servomagazine.com**). Be sure to have your AX-12+ controller hardware ready to roll as next time we're going to concentrate on the firmware. **SV**

Fred Eady can be reached via email at fred@edtp.com