Chapter 9

Hardware Communication

We have designed our robot controllers and robot hardware control node in the last chapter. Next, we need to send these commands to real robot hardware controllers. Theoretically, it does not belong to the category of ROS programming. However, it is still an essential part of your software architecture in the ROS frame. Let's discuss it in this chapter.

Given the diversity of robot hardware actuators and their interfaces, creating a single unified interface for all of them is impractical. Instead, by following a hierarchical (top-down) model - ranging from abstract, human-understandable commands down to concrete machine instructions or binary code - we can make our software more extensible and reusable across common layers. In designing such a model, we must strike a balance: it should not be overly complex, like the seven-layer `OSI` (Open System Interconnection) model used in networking, nor so simple that layers become tightly coupled, where a change in one layer adversely affects others.

In this chapter, we will take the `Hex7bot` hardware interface that the author has developed and explain some robot hardware communication design. Although it is not as good as you think, at least it provides a reference or some hints on your robot design.

9.1 Robot Hardware Architecture

Maybe you have learned it from chapter one: `Hex7bot` has a very simple hardware architecture (Figure 9.1):

The `Hex7bot` hardware controllers have a classical hierarchical structure as shown in the above figure. The host, in the configuration of Raspberry Pi 4B running Ubuntu 18.04 and `ROS Melodic` installed, communicates with the `Arduino Mega2560` board over a `USB-A` to Micro cable. Arduino board uses this USB connection with the host machine for both programming and serial communication. In Ubuntu OS, this USB device will be mapped to `/dev/ttyUSB0` serial device. Please note that by default, the device attribute is not fully controlled for regular users, you may need change device attributes so as to download new firmware and send new commands, etc.

```
sudo chmod 777 /dev/ttyUSB0
```

Or refer to the last chapter (Tips and Tricks) to change this device attributes permanently,

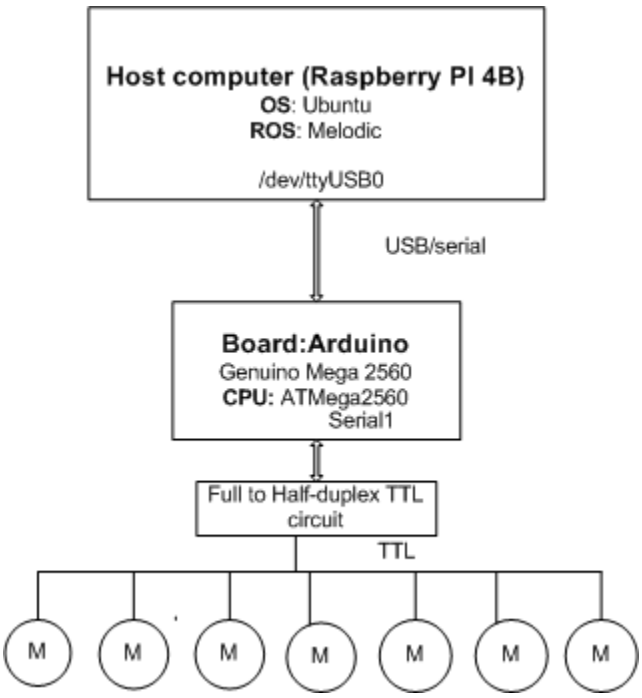

Figure 9.1: Structure of Hex7Bot Servo system

The customized Arduino mega2560 board (Figure 9.2) in the robot base in turn communicates with 7 digital `TTL` servo motors over a `half-duplex` , single-wire `TTL` bus. If you only have a similar development board, such as,

You need an extra convert board to convert `UART` (full-duplex `TTL`) signals into a half-duplex single-wire `TTL` bus, such as UART to half-duplex `TTL` converter - `URT-1` servo board (from the motor vendor, see Figure 9.8), Alternatively, you can create a simple passive circuit (Figure 9.3) to convert a full-duplex `TTL` serial port (TX/RX) signal into a half-duplex, single-wire `TTL` bus with one open-drain transistor.

This design works by using an open-drain `NPN` transistor to control the single shared data line, allowing a master device to transmit and receive on the same pin. The half-duplex converter relies on a pull-up resistor and a single `NPN` transistor (like S8050) to combine the TX and RX lines into one. Since there is no direction control pin, The `UART` protocol between all `TTL` devices and the master should follow 1 Start bit and 8 data bit and 1 stop bit and use exact data rates. A start bit (transition to low state logic 0) signals the beginning of a data frame and alerts receiver that a new data frame is arriving.

- The shared bus line (`TTL`), `TXD` and `RXD` are all kept high by the pull-up resistor when no device is transmitting.
- When the master needs to transmit, TXD line goes low. This low signal drives the transistor, which pulls the single TTL low, sending the data. If master transmits logic "1" by `TXD` line high, TTL is high by idle and NPN is off.
- When a slave device transmits, it pulls TTL low to send this data and `TXD` is pulled low when NPN is on. When TTL sends logic "1", the diode between TTL and TXD is forward biased, `TXD` is high. `NPN` is off due to negative base voltage V_B.
- The master receives data by monitoring TTL. When the TTL is low, the transistor is on, collector voltage is low because of the pull-up resistor. The master's own TX line must be held high during this time so it does not interfere.

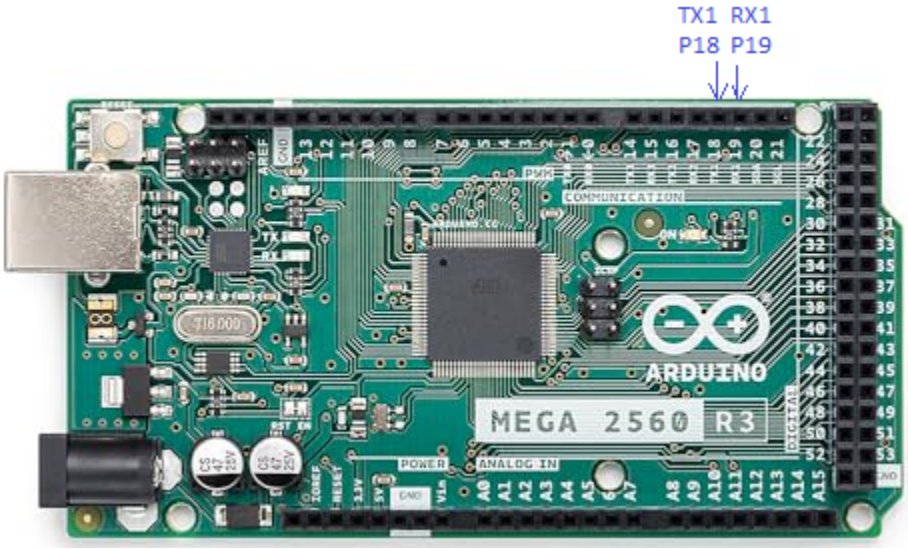

Figure 9.2: Arduino Mega 2560 Rev3

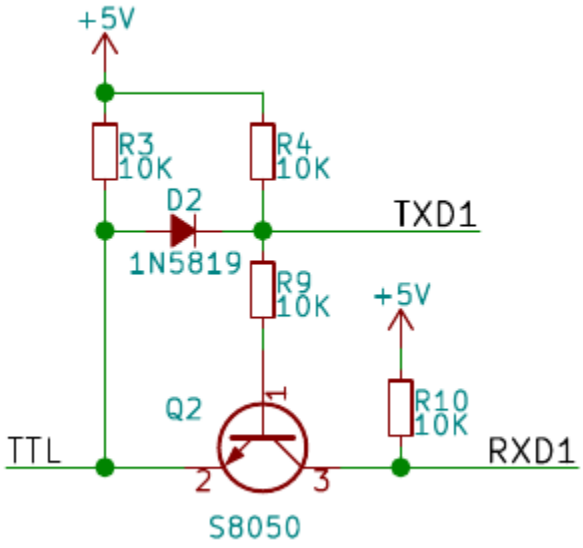

Figure 9.3: Circuit for UART to half-duplex TTL conversion

Note: - Don't confuse it with any `RS-485` servo motor, and never connect any `RS-485` servo motor in the `TTL` chain. `TTL` servo motors usually only have 3-wire pins (VDD/GND/TTL). However, RS-485 servo motors usually have 4-pins(VDD/GND/D+/D-). The RS485 interface allows you to control multiple motors over a single pair (D+/D-), providing high reliability in noisy environments. If your robot arm uses an RS485 interface, you will need a `UART-RS485` converter or controller to relay digital commands.

The firmware developed in Arduino IDE for `ATmeage256` receives commands from the host machine, such as position setpoints for every motor with its encoder counter, decomposes them into individual motor commands or combo commands, and unicasts/broadcasts them over the `Full-duplex` to the half-duplex TTL bus. The firmware of the Arduino board is also responsible for reading all current motor's encoder values, converting them back to joint state status messages and sending back to the host machine in two different mode: either upon `joint_state` request from host machine, or reporting status unsolicited.

Every servo motor on Hex7bot is a microcontroller-controlled intelligent module and can be controlled over a single-wire `TTL` bus. such as servo motors on joint 5, 6 and joint 7(i.e, gripper) which are `HLS3606M` (Figure 9.4: picture from the `Feetech` company), have the following characteristics:

Figure 9.4: Hex7Bot Servo motor

Please see Appendix B for detailed specifications.

Servo commands are based on the HDLC(high-level Data Link Control) protocol, follow the following format (Figure 9.5) :

Frame Header 0xFF 0xFF	Motor ID 0~253 255: broadcast	Length	Instruction	Parameters (1, 2, 3,...,n)	Check sum

Figure 9.5: motor servo protocol

We have discussed hardware control interface - ROS side design in the last chapter. It is the time to discuss hardware side's design.

9.2 Robot Hardware Driver.

In the 'hardware_interface' node, there are several members for robot control:

- IpcHandle * jntCmdChan;
- Session *hex7bot_sess_ ;
- HEX7BOT_RBT hex7bot_robot_;
- Hex7BotDriver * hex7bot_driver_ ; //hardware adapter

We have learned from previous chapters that `HEXBOT_RBT` is a robot model class and is responsible for robot kinematics. What about `IpcHandle` , `Session` and `Hex7BotDriver` ? Hex7BotDriver is a Hex7Bot hardware adapter class which is derived from the hardware adapter class (abstract class) `robot_driver` :

robot_driver class header file

```
class RobotDriver
{
public:
   RobotDriver(Session *sessIO);
   virtual ~RobotDriver();

   virtual int signal (int sig_id) ;
   virtual int probe() ;
   virtual int attach() ;
   virtual int detach();
   virtual int read (unsigned char *dataBuf, int bufSize);
   virtual int write (const unsigned char *dataBuf, int bufSize);
   virtual int ioctrl(uint32_t sub_cmd, unsigned char *dataBuf,
            int bufSize) = 0;

protected:
   Session * m_sess;
   int m_rbtState;
};
```

Robot_driver class has an important member - `Session` that controls how messages are sent, i.e., `IPC` , socket or serial port etc. it also defines standard interfaces like regular computer device driver:

- `attach()` : Open and connect the hardware
- `detach()` : Disconnect/power off
- `signal()` : Signal hardware with signal number: RCCL code for puma260
- `probe()` : Probe hardware status
- `read()` : Read robot status data frames from hardware
- `write()` : Write robot command data frames to hardware
- `ioctrl()` : Special commands to modify robot mode etc (pure virtual function), derived class has to define it.

the `Robot_driver` class has an important composition: the pointer to a `Session` type object : It is responsible for converting high-level commands to lower-level machine commands. As its name hints, it acts on the session layer of OSI model by creating and maintaining an active session connection between the ROS environment and robot hardware.

Different robot types will have different hardware drivers instances although they are all derived from same

base class, as shown in Figure 9.6,

When the session receives the command/data from the higher layer, it will wrap the command/data with its session header, such as:

```
#pragma pack(1)
typedef struct SESSION_HEADER_t
{   uint32_t sync;
    uint32_t time_sec;
    uint32_t time_nsec;
    uint32_t sn; //serial number
    int32_t src;  //source module id
    int32_t dst; // dest module id
    int32_t type; //messageType_e
    uint32_t size; // Message header including header (at least 32 bytes)
} SESSION_HEADER;

#pragma pack() /* restore default pack value */
```

The `Session` class is the common interface for all robot drivers but initialized with different configurations. When creating a detailed hardware interface, you need to tell how your `Session` object is constructed, such as,

```
jntCmdChan = new IpcHandle(ARM_7BOT_SERIAL_TRX, hex7bot_conf_file.c_str());
ROS_INFO(" HEX7BOT Joint control interface file: %s",
hex7bot_conf_file.c_str());

hex7bot_sess_ = new Session(jntCmdChan, ARM_7BOT_SERIAL_TRX,
    ARM_7BOT_ARDUINO_TRX, IPC_SERIAL_PORT);
```

Where constructor Session::Session(IpcHandle *ipcHdl, uint32_t src, uint32_t dst, uint32_t type) is called to indicate a command channel between hardware module: `PUMA260_HOST_IO_TRX` and `ARM_7BOT_ARDUINO_TRX` is established over the serial port. It is a duplex data/status channel

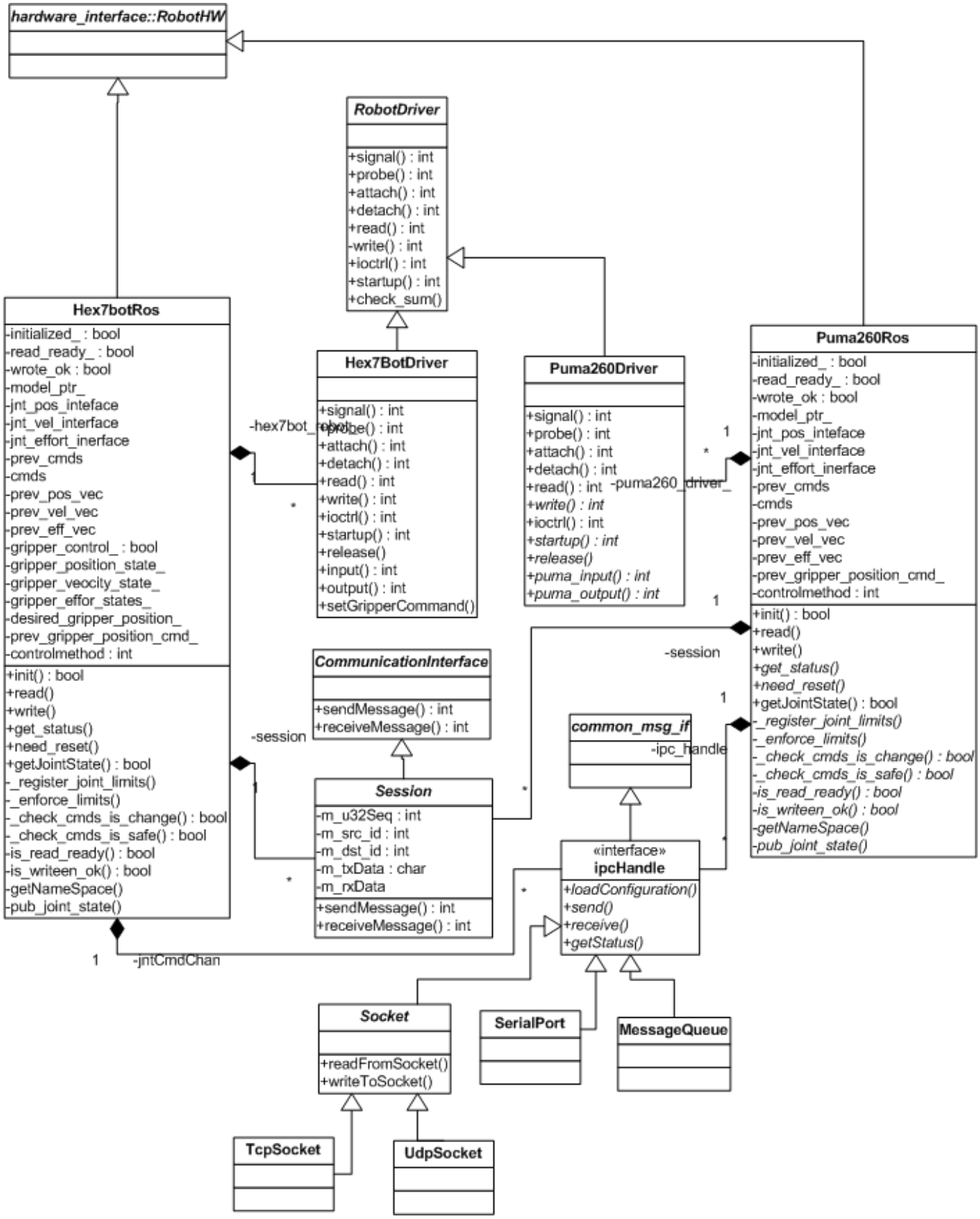

Figure 9.6: Robot driver hierarchical structure

```
enum IpcType_e
{
   IPC_QUEUE,
   IPC_TCP_SOCKET,
   IPC_UDP_SOCKET,
   IPC_SERIAL_PORT,
   TOTAL_IPC_TYPES
};

//name in MODULE_TYPE_STR.
enum ModuleType_e
{
  /*0*/   ROS1_HOST = 0,  /* ROS1 environment*/
  /*1*/   JOY_STICK = 1,
  /*2*/   PUMA260_HOST_IO_TRX = 2,  /* PUMA260 Host Control channel*/
  /*3*/   PUMA260_RBT_IO_SVR  = 3,  /* PUMA260 robot state channel */
  /*4*/   PUMA260_HOST_JNT_CMD_TX = 4, /*PUMA260 HOST Position control channel:
                                        commands: osrf_msgs/JointCommands */
  /*5*/   PUMA260_HOST_JSTATE_RX = 5, /* Puma260 Joint state channel */
  /*6*/   PUMA260_HOST_JTRAJ_TX =   6, /* PUM260 Joint trajectory command*/
  /*7*/   ARM_7BOT_SERIAL_TRX  = 7,
  /*8*/   ARM_7BOT_ARDUINO_TRX =   8,
  /*9*/   LastModuleTypes =  9
};

const static ModuleType INVALID_MODULE = -1;
typedef list<ModuleType> ModulelList_t;

static const char* const MODULE_TYPE_STR[LastModuleTypes] =
{
   "ROS1_HOST",
   "JOY_STICK",
   "PUMA260_HOST_IO_TRX",
   "PUMA260_RBT_IO_SVR",
   "PUMA260_HOST_JNT_CMD_TX",
   "PUMA260_HOST_JSTATE_RX",
   "PUMA260_HOST_JTRAJ_TX",

   "ARM_7BOT_SERIAL_TRX",
   "ARM_7BOT_ARDUINO_TRX",
};
```

The Session object can be also be initialized by a configuration file such as: ipc_msg.conf

```
# module_name | direction | destination module(enum) | interface type |
        device handle (address) | port number

PUMA260_HOST_IO_TRX     CLIENT_TRX
        PUMA260_RBT_IO_SVR TCP 192.168.1.175 15030
PUMA260_HOST_JNT_CMD_TX CLIENT_TRX
        PUMA260_RBT_IO_SVR TCP 192.168.2.35 15030
```

Currently, the session class supports `POSIX` message queue (OS layer), TCP, UDP and serial port connections. Unless you have a new hardware type, you seldom need to update/extend the session class which is defined in *ipc_common* library. Most of time, you just need include its header and link *ipc_common* library for your application.

In addition, *ip_common* library is independent of ROS and was developed without using any ROS resources.

9.3 Hardware Message Frame

There are an intimidate layer(header) between session header and hardware command.

such as:

```
typedef struct APP_CMD_HEADER_tag
{
  uint8_t   u8Flag;  /*Must be 0x82*/
  uint8_t   u8Proto; /* protocol type */
  uint8_t   u8Seq;  /*Command Sequence */
  uint8_t   u8Pad;   /*command data frame padding: 0-3 bytes*/
  uint16_t  u16Len;  /* Command Length in half Dword */
  uint8_t   u8Cmd;   /* LSB first 00:command*/
  uint8_t   u8CmdExt;
  /* note:Command length is body length, not including
    terminator 0x0D 0x0A*/
} APP_CMD_HEADER;
```

The intermediate layer mimics the network/MAC layer of the OSI, can send the same command to one (unicast) or multiple (multicast or broacast) device nodes. Another application scenario is for real network adapter device e.g. `FPGA/SoPC` (System-on-a-Programmable-Chip) logic between the host and robot controller and can split the large data to smaller data chunks (Figure 9.7).

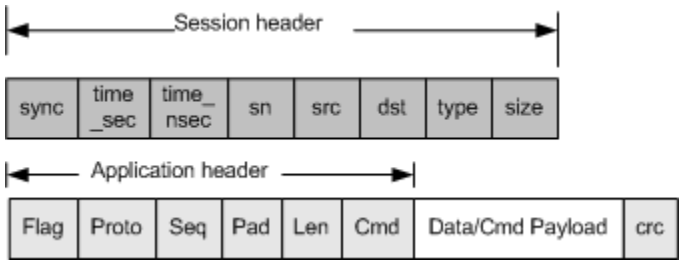

Figure 9.7: data frame structure

If we enable our data flow trace in hex format, you will see some outputs like:

```
Read all servo positions[Encoder/counts]:
 SerialPort sent - Length=52
 5A FA FA 5A DB 9A 7A 68 E0 4E 00 0D 2E 00 00 00| Z  Z  zh N  .
 07 00 00 00 08 00 00 00 03 00 00 00 34 00 00 00|            4
 82 01 2E 00 08 00 02 00 FF FF 06 04 02 3C 02 B5|  .          <
 14 22 0D 0A                                    | "
 SerialPort receive - Length=32
 5A FA FA 5A 0E 00 00 00 00 21 38 15 2D 00 00 00| Z  Z     !8 -
 08 00 00 00 00 00 00 00 08 00 00 00 34 00 00 00|            4
 SerialPort receive - Length=20
 82 01 2D 00 08 00 02 00 FF FF 06 04 00 34 00 C1|  -          4
 FF FF 0A 0A                                    |

[ INFO] [1752865499.278524557]: Hex7botRos:: HW read
Hex7botRos:: HW read (degree):
 J1=-4.47257 J2=54.6484 J3=-3.55452 J4=-0.703134 J5=3.1641 J6=-0.0878917
 SerialPort sent - Length=52
 5A FA FA 5A DB 9A 7A 68 08 B7 9E 10 2F 00 00 00| Z  Z  zh    /
 07 00 00 00 08 00 00 00 03 00 00 00 34 00 00 00|            4
 82 01 2F 00 08 00 02 00 FF FF 00 04 02 38 02 BF|  /          8
 3D 86 0D 0A                                    |=
 SerialPort receive - Length=32
 5A FA FA 5A 0E 00 00 00 00 A8 CB 18 2E 00 00 00| Z  Z        .
 08 00 00 00 00 00 00 00 08 00 00 00 34 00 00 00|            4
 SerialPort receive - Length=20
 82 01 2E 00 08 00 02 00 FF F5 00 04 00 05 08 EE|  .
 FF FF 0A 0A
```

You will see `0x5AFAFA5A` - Sync flag for Session header and application frame start with 0x82. where motor command in this sample "FF FF 00 04 02 38 02 BF 3D" meaning: - FF FF: frame header - 00: read motor id=0 (joint 1)
- 04: payload 4 - 02 : Read Instruction - 38: Position register address 0x38 - 02: register data length - BF 3D: check sum from FF FF to register data length

0D 0A: is application frame (from 0x82), always 0xD 0A

9.4 Firmware Design

Based on `Hex7bot` hardware architecture, we have known the Arduino board's role (servo processor) in the system: Listen to its host machine for servo commands, execute the commands and at the same time collect and report present motors' states, which makes the firmware much easier since there are no floating-point computations involved.

Perhaps many readers don't have much experience in embedded software development or never touch a micro-controller before. The good thing is that Arduino has provided us an easy-to-use IDE environment and well design software framework.

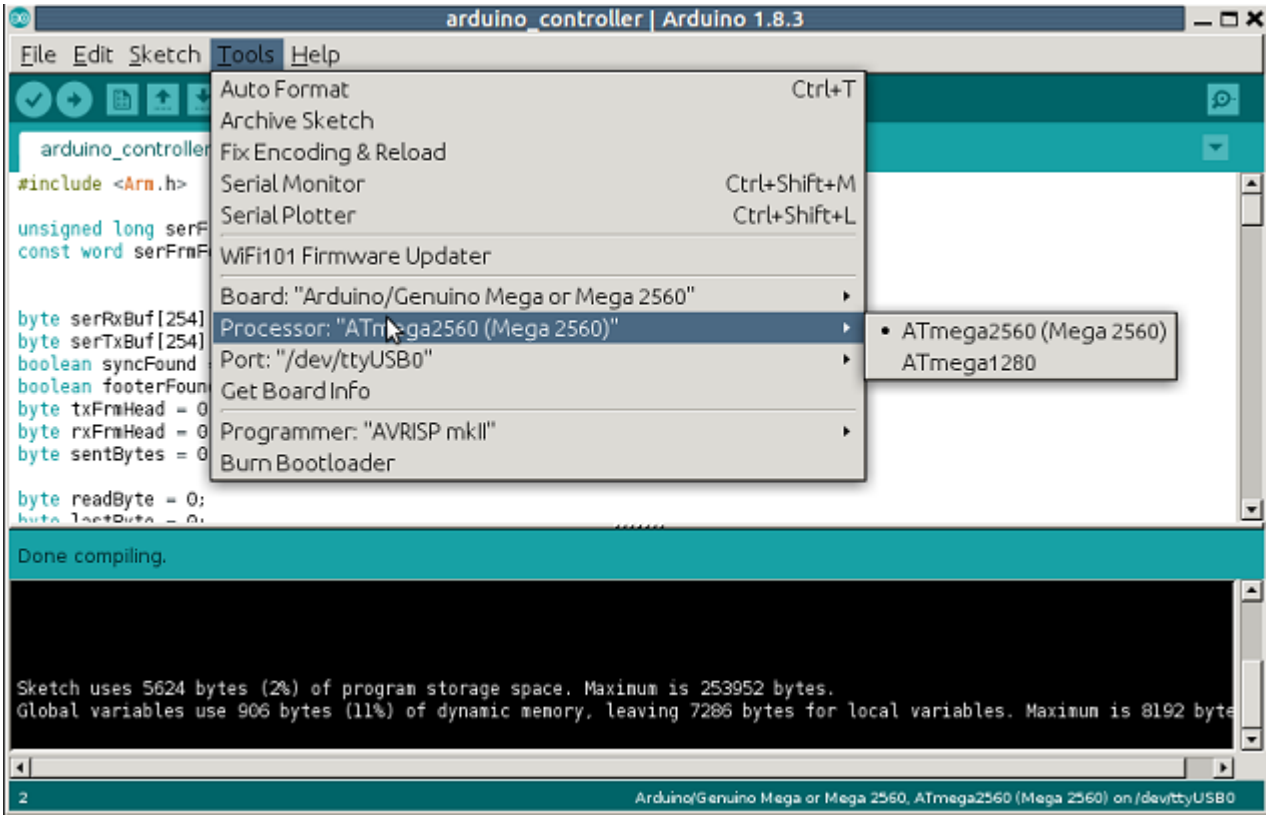

Figure 9.8: Arduino IDE environment

Arduino *http://www.arduino.cc* is an open-source electronics platform based on easy-to-use hardware and software.

The Arduino IDE and programming language are designed to be beginner-friendly, making it easier for those without extensive programming or electronics backgrounds.

The Arduino provides an easy-to-use software framework for beginners by hiding most micro-controller programming details such as timer, interrupt, and bootloader. The user usually just needs to implement two functions (in Arduino term: sketch) for a typical task: such as - `setup()` and - `loop()`

The Arduino is also open source and extensible software frame and its source tool or c++ libraries are also published as source code.

In addition, Arduino device does not require special JTAG programmer. you can program and debug supported micro-controllers through its debugging interfaces. There are extensible inexpensive development boards available online stores for hobbyists.

9.4.1 Setup Function

```
#include <Arm.h>

unsigned long serFrmStartDword = 0x5AFAFA5A;
const word serFrmFooterWord =   0x0A0D; // 0D 0A

byte serRxBuf[254];
byte serTxBuf[254];
boolean syncFound = false;
boolean footerFound = false;
byte txFrmHead = 0;
byte rxFrmHead = 0;
byte sentBytes = 0;

byte readByte = 0;
byte lastByte = 0;
unsigned long readDWord = 0; // MSB read first
unsigned long sn = 0;
int timeout = 1000; // sets timeout to wait 1 second for a number
long lastTime = 0;

boolean receivedFrame = false;
boolean transmitFrame = false;

void setup() {
  MyArm.begin(USB_SER);
  Serial.begin(115200); // begin serial @ 115200 bits/sec
  Serial1.setTimeout(10);

  //initialize
  MyArm.position_init();
  delay(1000);

}
```

9.4.2 Loop Function

```
void loop() {
   if (Serial.available()) {
      readByte = Serial.read();

      if (!syncFound) {
          rxFrmHead = 0;
          readDWord <<= 8; //Shift 8 bit
          readDWord += readByte;
          if (readDWord == serFrmStartDword) {
             syncFound = true;
             memcpy(serRxBuf, &serFrmStartDword, sizeof (unsigned long));
             rxFrmHead = 4;
          }

      }
      else if (syncFound == true && rxFrmHead < 254) {
        lastByte = serRxBuf[rxFrmHead-1];
        serRxBuf[rxFrmHead++] = readByte;

        if (lastByte == 0x0D && readByte == 0x0A) {
           receivedFrame = true;
           //get one complete Frame from the host machine
           decode(serRxBuf, rxFrmHead);
           //post process
           syncFound = false;
           rxFrmHead = 0;
           readDWord = 0;
        }
      }
  }
  else {
       hostSend();
  }
}
```

And local functions :

- decode()

```
void decode(byte *data, byte length) {
    //respBuf[4] : status
    byte respBuf[6] = {0xff, 0xf5, 0xff, 0x06, 05, 0};

   if (length <= 40) {
      respBuf[4] = 0x04 ; //not enough header size
      respBuf[5] = (byte)((~(respBuf[2] + respBuf[3] + respBuf[4]))
                   & (0xff));
      sendServoDataToHost(respBuf, 6);
      return ;
   }

   byte command = data[38];
   byte cmd_length = data[36];

   switch (command) {
     //sync write command
     case 0x03: //write data
     {
         arm_sync_write(&data[40], cmd_length);
         Serial1.flush();
         break;
     }
     case 0x02: //read data
     {
         //decode id , regStartAddr, data, and readlen
         servo_read(&data[40], cmd_length);
         break;
     }
     case 0x01: // ping command
     { //loopback write
         servo_ping(&data[40], cmd_length);
         break;
     }
     default:
     {
        respBuf[4] = 0x07 ; //not engout header size
        respBuf[5] = (byte)((~(respBuf[2] + respBuf[3] + respBuf[4]))
                     & (0xff));
        transmitFrame = true;
        sendServoDataToHost(respBuf, 6);
        break;
     }
   }
}
```

local function hostSend()

```
void hostSend()
{  sentBytes = 0;
   while (transmitFrame == true && sentBytes < txFrmHead) {
      //delay(10); // 10 ms delay here
      size_t availableWrittenBytes = Serial.availableForWrite();
      byte currSend = (txFrmHead - sentBytes) > availableWrittenBytes ?
                  availableWrittenBytes : (txFrmHead - sentBytes);

      byte currSentBytes =  Serial.write(serTxBuf + sentBytes, currSend);
      sentBytes += currSentBytes;
      if (sentBytes == txFrmHead) {
          transmitFrame = false;
          txFrmHead = 0;
          sentBytes = 0;
      }
   }
}
```

- arm_sync_write

```
boolean arm_sync_write(byte *data, byte data_len) {
   byte i;
   byte offset;
   boolean ret = false;
   word position;
   word runtime;

   if (data_len < 11){
       return ret;
   }
   offset = 3 ; // total length
   byte sync_total_len = data[offset++];
   byte cmd = data[offset++]; //offset=4
   byte regAddr = data[offset++]; //offset=5
   byte one_datalen= data[offset++] + 1; //ofset=6

   if (cmd != INSTRUCTION_SYNC_WRITE || sync_total_len < 9 ||
           one_datalen != 5){
       return ret ;
   }

   for (i = data[offset]; offset < sync_total_len && i < MyArm.Steer_Num;
           offset += one_datalen, i=data[offset]){
       //default data order is always littleEndian from host machine
       position = (data[offset + 2] << 8) + data[offset + 1];
       runtime = (data[offset + 4] << 8) + data[offset + 3];

       if (i < MyArm.Steer_Num){
           MyArm.joint_move(i, position , runtime);
           ret = true;
       }
   }
   return true;
}
```

- servo_read

```
boolean  servo_read(byte *data, byte data_len){
   boolean ret = false;
   byte sentBytes = 0;
   byte id = data[2];
   byte readSer1Byte = 0;
   byte cnt = 0;
   byte retry1 = 0;
   byte retry2 = 0;
   byte readbuf[8] = { 0xFF, 0xF5, id, 4, 0x01, 0, 0,
           (byte)((~(id+5)) & (0xff)) };

   while (cnt == 0 && retry1 ++ < 2){
      while(Serial1.availableForWrite() < data_len);

      Serial1.write(data , data_len);
      Serial1.flush();
      delay(10);
      while (Serial1.available() < 8 && retry2++ < 4){
          delay(20); // wait for 20ms here
      }

      while (Serial1.available()){
        // cnt = Serial1.readBytes(readbuf , 8);
        readSer1Byte = Serial1.read();
        readbuf[cnt++] = readSer1Byte;
        delay(2);
     }
   }

   if (cnt == 0){
       readbuf[2] = id;
       readbuf[3] = 4;
       readbuf[4] = 0x01; // Error , not connected
       readbuf[5] = 0;
       readbuf[6] = 0;
       readbuf[7] = (byte)((~(id+5)) & (0xff));
       cnt = 8;
   }
```

```
    //if the servo is big endian based, we need swapped the
    // payload for host.
    if (!MyArm.steer[id]->littleEndian){
        readSer1Byte = readbuf[5];
        readbuf[5] = readbuf[6];
        readbuf[6] = readSer1Byte;
    }

    sentBytes = (byte) sendServoDataToHost(readbuf, cnt);
    if (sentBytes == cnt){
      ret = true;
    }
    return ret;
}
```

- servo_ping

```
boolean servo_ping(byte *data, byte data_len)
{
    byte readSer1Byte;
    boolean ret = false;
    byte id = data[2];
    byte sentBytes = 0;
    byte retry1 = 0;
    byte retry2 = 0;
    byte maxTries = 2;
    byte buf[6] = {0xff , 0xf5,
                   id ,
                   0x06 , 0x01, (byte)((~(id+7)) & (0xff))};
    byte cnt = 0;

    while (cnt == 0 && retry1++ < maxTries) {
       while(Serial1.availableForWrite() < data_len);
       Serial1.flush();
       while (Serial1.available()) { Serial1.read();}
       Serial1.write(data, data_len);

       while (Serial1.available() < 6 && retry2++ < 4){
           delay(10); // wait for 10ms here
       }
      while (Serial1.available()){
       // Serial1.readBytes(buf , 6);
        readSer1Byte = Serial1.read();
        buf[cnt++] = readSer1Byte;
        delay(2);
      }
    }
```

```
    //error case
    if (cnt == 0) {
      buf[4] =   0x05; // no response
      buf[5] = (byte)((~(buf[4]+ 6 + id)) & (0xff));
      cnt = 6;
    }

    sentBytes = (byte) sendServoDataToHost(buf, cnt);

    if (sentBytes == cnt) {
      ret = true;
    }

    return ret;
}
```

- sendServoDataToHost: Send sevo data to host

```
byte sendServoDataToHost(byte *dataBuf, byte datalen){
    byte u8pad = 0;
    byte i;

    unsigned long time_s = millis() /1000;
    unsigned long time_ns = (millis() - time_s *1000) * 1000000;

    if ((datalen & 0x03)> 0){
        u8pad = 4 - (datalen & 0x03);
    }
    //compose frame
    memcpy(serTxBuf, &serFrmStartDword, sizeof(unsigned long));
    txFrmHead = 4;
    memcpy(serTxBuf + txFrmHead, &time_s, sizeof(unsigned long));
    txFrmHead += 4;
    memcpy(serTxBuf + txFrmHead, &time_ns, sizeof(unsigned long));
    sn += 1;
    txFrmHead += 4;
    memcpy(serTxBuf + txFrmHead, &sn, sizeof(unsigned long));
    unsigned long src = 8;
    unsigned long dst = 0;
    txFrmHead += 4;
    memcpy(serTxBuf + txFrmHead, &src, sizeof(unsigned long));
    txFrmHead += 4;
    memcpy(serTxBuf + txFrmHead, &dst, sizeof(unsigned long));
    unsigned long type = 8;
    txFrmHead += 4;
    memcpy(serTxBuf + txFrmHead, &type, sizeof(unsigned long));
    txFrmHead += 4;
    memset(serTxBuf + txFrmHead, 0, sizeof(unsigned long));
    txFrmHead += 4;
```

```
    serTxBuf[txFrmHead++] = 0x82; //'$'
    serTxBuf[txFrmHead++] = 0x01; // version 0.2
    serTxBuf[txFrmHead++] = sn & 0xFF; //cmmand sequence
    serTxBuf[txFrmHead++] = u8pad;
    serTxBuf[txFrmHead++] = datalen;
    serTxBuf[txFrmHead++] = 0; //LSB
    serTxBuf[txFrmHead++] = 0x02; // Report
    serTxBuf[txFrmHead++] = 0;

    memcpy(&serTxBuf[txFrmHead], dataBuf, datalen);
    txFrmHead += datalen;
    serTxBuf[txFrmHead++] = 0xFF; //ignore Session checksum
    serTxBuf[txFrmHead++] = 0xFF; //ignore session checksum

    for (i=0; i < u8pad; i++){
      serTxBuf[txFrmHead++] = 0xaa ;//padding
    }

    serTxBuf[txFrmHead++] = 0x0D;
    serTxBuf[txFrmHead++] = 0x0A;

    serTxBuf[28] = txFrmHead;

    sentBytes = 0;
    transmitFrame = true;

  return txFrmHead;
}
```

Please note: This sketch program depends on two libraries: `ARM` and `Steer_Protocol` that are originally from the `7-bot` project and `Arduino` IDE's libraries. It won't compile without two libraries.

The two libraries define all servo commands/instructors, such as,

```
//broadcast address
#define BROADCAST_ADDR                 0xFE

//the macros below is instructions
#define INSTRUCTION_PING               0x01
#define INSTRUCTION_READ_DATA          0x02
#define INSTRUCTION_WRITE_DATA         0x03
#define INSTRUCTION_REG_WRITE          0x04
#define INSTRUCTION_ACTION             0x05
#define INSTRUCTION_RESET              0x06
#define INSTRUCTION_SYNC_WRITE         0x83


//the macros below is storaged in EEPROM
// Id address: range (0~253)
#define ID_REG                      0x05
#define MIN_ANGLE_LIMIT_H              0x09
#define MIN_ANGLE_LIMIT_L              0x0A
#define MAX_ANGLE_LIMIT_H              0x0B
#define MAX_ANGLE_LIMIT_L              0x0C
#define MAX_TORQUE_H                   0x10
#define MAX_TORQUE_L                   0x11
#define DEFAULT_SPEED                  0x12
#define MIDDLE_POSITION_H              0x14
#define MIDDLE_POSITION_L              0x15

//the macros below is storaged in RAM
// 1:open; 0: close
#define TORQUE_SWITCH                  0x28
#define TARGET_POSITION_H              0x2A
#define TARGET_POSITION_L              0x2B
#define CURRENT_POSITION_H             0x38  //Read only
#define CURRENT_POSITION_L             0x39  //Read only
#define TARGET_SPEED                  0x41

//Write mode
//Asynchronous
#define REG_WRITE                      0x01
//Syncronous
#define SYNC_WRITE                     0x02
```

For more detail information about motor registers, please refer to Appendix B.

9.4.3 Debugging and Troubleshooting

Debugging and troubleshooting firmware is a little challenging, especially when the `Arduino` platform does not provide a `JTAG` programmer and the debugging serial port has been used for host-client communication. In addition, the serial port with the host machine has been used as a command/status bus and can not dump tracing information. Two alternative solutions for it:

1. Loop back serial port TX and RX pins and make sure the commands sent from the host are able to echo back.
2. Use a serial-port analyzer like the `Saleae` logical analyzer, which supports extended high-layer analyzer (software plugin developed by the Python) and probe the one-wire TTL bus.
3. Print serial port traffic with hexadecimal format (verbos mde) in the code, like previous section logs.
4. Connect the motor on the robot over the URT-1 board (Figure 9.9) directly and control the servo motor to a predefined point and read the servo motor's encoder value with the motor troubleshooting tool, such as FT SCservo Debug tool provided by motor vendor.

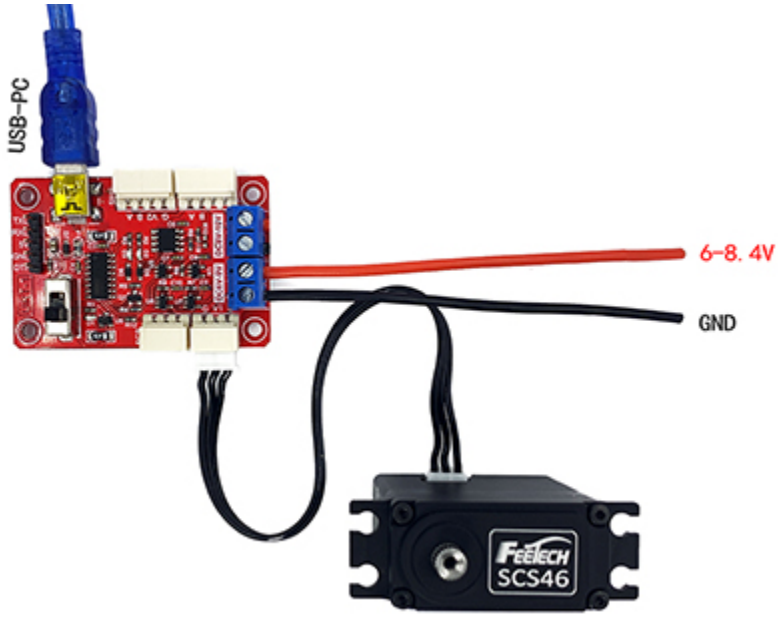

Figure 9.9: URT-1 motor driver board

and Servo motor debug/config tool (Figure 9.10):

9.5 Summary

In this chapter, we discussed how to design a robot's hardware interfaces with the Hex7Bot as an example, i.e., fill the gap between `hardware_interface_node` (ROS) and the robot's actuators/motors, which are usually out of ROS scope.

We've learned that software architecture for it has been optimized for interface standardization and code reuse by encapsulating all common functions/interfaces to libraries (`session`) and providing a common interface (`robot_driver`)for future new robot types.

In addition, we briefly explored how to design and develop a robot controller's firmware. Since the host machine handles high-performance computations, the task of the lower-level board is much simpler: it receives and executes servo commands (i.e., encoder counter) from the host and report current position status. Finally, we also mentioned some debugging and troubleshooting skills and utility tools for readers interested in firmware development.

So far, we have a basic understanding of the robot's entire software system except for offline simulation (next chapter's topic). However, it does not yet involve software modules at the sensing, perception, and task

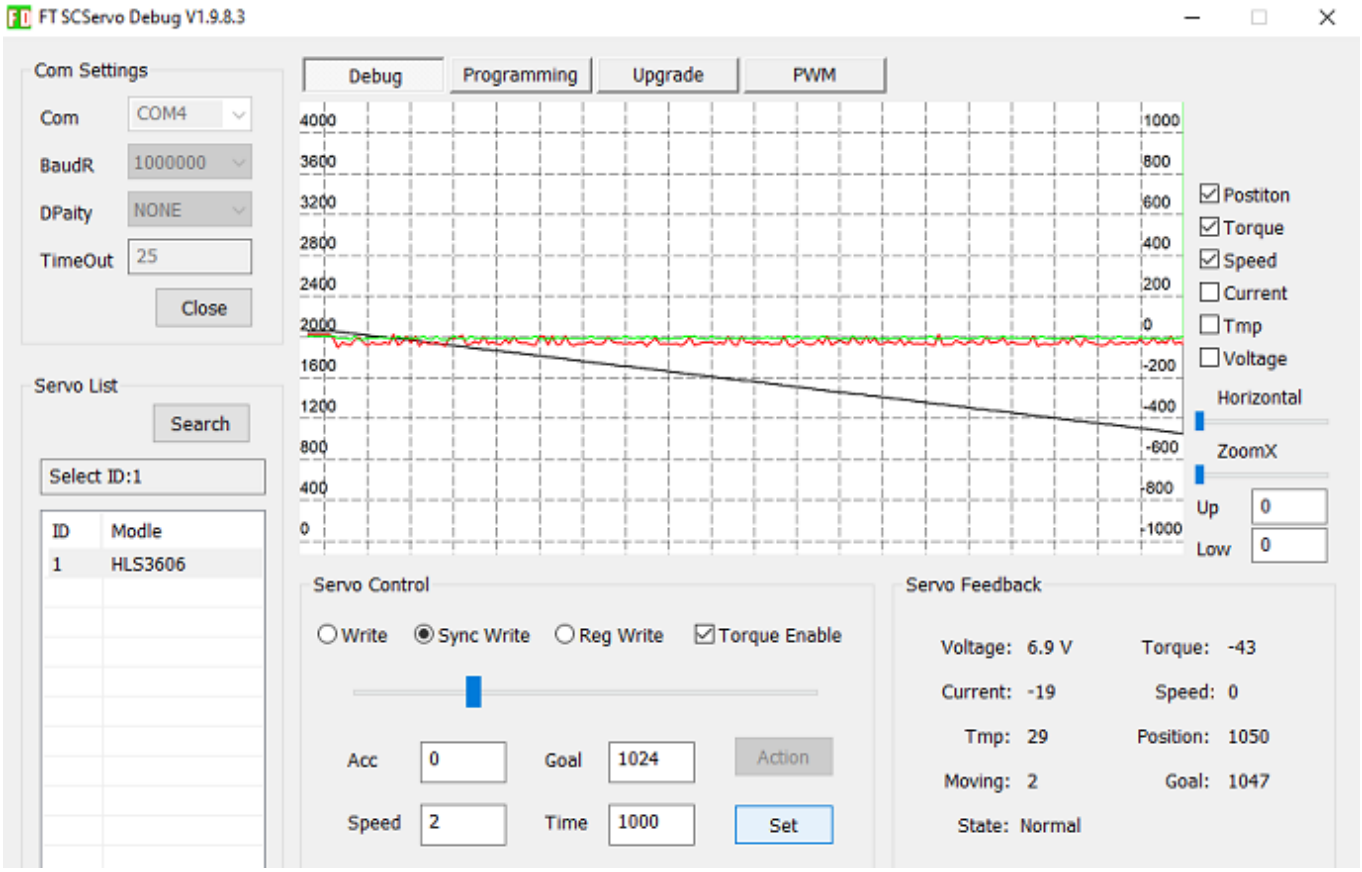

Figure 9.10: FT SCServo Debug tool