B Flexible eXperiments (FX) Sketch for Arduino UNO

As you may observe while reading this book, the Arduino sketch development gradually grows. That means a sketch in later chapter generally maintains certain features from previous ones, except some that requires specific hardware setup such as state feedback. Eventually, `ex8_7.ino` at the end of chapter 8 contains all functions used in this book. It deserves to have a name. So I just call it Flexible eXperiments, or FX in short.

Due to limited resources of the Arduino R3, I cannot make FX to become as user friendly as I wish. The sketch consumes 69% of program storage space and 87% of dynamic memory of the AVR microcontroller. After compiling, you'd see a warning "Low memory available, stability problems may occur," though I have not yet spotted any problem.

The reader can use FX in any application that seems fit, or modify it to suit your needs. There is no license or royalty of any kind. This author, of course, shall not be responsible for any damage or undesirable outcome resulting from using this sketch.

I have to say it here. Arduino is great for educational purpose. If you need a controller for real feedback control applications, with stringent stability and performance criteria, better design your own board with a 16 or 32 bit microcontroller. The concepts and knowledge learnt from this book will surely be applied to a more advanced embedded system.

Main Features

- Control scheme can be switched between PID and custom-designed, output feedback controller
- PID controller fully equipped with anti-windup, setpoint and derivative weights, and autotuning function.
- PRBS generator for offliine least-square system identification
- Selectable feedback input between voltage (ADC pin A2) and encoder (I2C at address 0x30)
- Selectable output between PWM only (pin 9) and PWM (pin 9) with direction output (pin 8)
- ADC moving average function for noise filtering
- Command interpreter for system and controller parameters setup via serial communication

FX Commands

We summarize below FX commands for parameter setup and operations. The command and argument are separated by a white space. If no argument is passed, the command returns the current value of that parameter.

Command Syntax	Description
`adma <on/off>`	turn ADC moving average on/off
`autotune`	start PID autotuning
`close`	close feedback loop
`controller <pid/cc>`	switch controller between PID and custom design
`datasize <value>`	set number of data points to <value>
`dirsense <0/1>`	set direction sense of motor (invert logic of pin 8)
`enc <value>`	set encoder to <value>
`fbloop`	check loop status (open or close)
`insel <adc/enc>`	select input (ADC pin A2, or encoder via I2C)
`kp <value>`	set proportional gain of PID controller
`ki <value>`	set integral gain of PID controller
`kd <value>`	set derivative gain of PID controller
`kt <value>`	set back calculation gain (anti-windup) of PID
`lsid`	excite plant with PRBS input and send data to host
`open`	open feedback loop
`outsel <pwm/pwmdir>`	select output (PWM only or PWM and DIR)
`pwm <value>`	set PWM to <value> (between +/- 100 %)
`step <value>`	step command to <value>
`verbose <on/off>`	set the amount of information returned
`wp <value>`	set proportional weight of PID controller
`wd <value>`	set derivative weight of PID controller
`y`	acquire current y value

Listing B.1 below is the complete FX sketch. You can also download it from the book's website. Don't forget to install timer2 library if you have not done so.

```
// FX.ino : Flexible eXperiments sketch
// designed for Arduino UNO R3
// by dew.ninja  September 2016

// This sketch is the final development
// from the book "Feedback Control with Scilab and Arduino."
// It implements PID and custom output feedback controller
// with configurable input/output, and supported functions
// such as PID autotune and PRBS generator for
// offline Least-square identification.
// See the book for more details.

#include <MsTimer2.h>
#include <Wire.h>

#define ADCIN 0    // A/D input
#define ENCIN 1    // I2C inp;ut
#define PWMOUT 0    // PWM only output
#define PWMDIROUT 1  // PWM/DIR output
#define PID 0      // controller = PID
#define CC 1      // controler = CC (custom controller)

const int PWMOut = 9;  // PWM signal
const int DIROut = 8;  // DIR signal
const int VO = A2;    // output of RC network
const int VCMD = A1;  // analog command from Pot
const int DATASIZEMAX = 5000;   // maximum data size
const float enc2deg = 0.075;  // convert encoder unit to degree by 360/4800

// voltage range
const float VMAX = 5;
const float VMID = 2.5;
const float VMIN = 0;

// PWM and ADC ranges
const int PWMMAX = 255;  // 8-bit PWM
const int PWMMID = 127;
```

```
const int PWMMIN = 0;
const int ADCMAX = 1023;   // 10-bit ADC
const int ADCMID = 511;
const int ADCMIN = 0;

const float RELAYOUT = 20;    // relay amplitude (in volt or PWM raw unit)
// controller parameter limits
const float KPMAX = 100000;  // proportional gain
const float KIMAX = 100000;  // integral gain
const float KDMAX = 100000;  // derivative gain
const float KTMAX = 100;
const float WPMAX = 1;
const float WDMAX = 1;

// autotuning limits
const float MAXATTIME = 10;  // maximum time to quit if no oscillation

float AD2V = 0;    // convert ADC unit to volt
float V2PWM = 0;    // convert volt to PWM unit
float outscale = 1;    // output scaling
int i = 0;          // index
String rcvdstring;   // string received from serial
String cmdstring;   // command part of received string
String  parmstring; // parameter part of received string
int noparm = 0;    // flag if no parameter is passed
int parmvalint;        // parameter value
float parmvalfloat;
int newcmd = 0;     // flag when new command received
int sepIndex;       // index of seperator
int datasize = 200;  //500;
float cmd = 0;          // command value
float cmdold = 0;     // previous value of command
float y = 0;         // output voltage value
float ev = 0;         // error variable
int pwmout = 0;       // pwm value
int dir = 0;          // direction
int dirsense = 0;      // direction sense to be XORed with dir
```

```
int datacapt = 0;    // flag for change in command value
int loopclose = 0;      // loop status flag 0 = open, 1 = close
int verboseflag = 1;    // response verbosity
int inputselect = ENCIN;  // input selection defaulted to ADC
int outputselect = PWMDIROUT;  // output selection defaulted to PWM only
float T = 0.01;    // sampling period
unsigned long T_ms;          // sampling period in milliseconds

// controller variables
// custom control coefficients
int controltype = PID;    // controller type 0 = PID, 1 = CC

int ix = 0;      // state index

// custom controller coefficients.
// this example is for DC motor control, with T = 0.01
int ccnumber = 7;    //number of coefficients
float   a[7]   =   {1,      -0.429281206809269840,      -0.947394212580144330,
0.169157376179709630,
0.383851810784896390,  -0.034325731432093165,  -0.141487138770148820};
float   b[7]   =   {   13.564640424971685000,      69.252296538840568000,   -
223.554070414441440000,
 162.957572108037970000,     33.899472575614986000,  -84.854729554906072000,
28.735085757699142000};
float x[7] = {0,0,0,0,0,0,0};   // controller states

// PID parameters
float kp = 1;    // proportional control
float ki = 0;    // integral
float kd = 0;    // derivative
float kt = 0;    // back calculation gain
float wp = 1;    // proportional weight
float wd = 1;    // derivative weight
float N = 20;    // D filter coeffieient

// controller coefficients as functions of PID parameters
float a1,a2;
float b1, b2, b3;
```

```
float c1, c2, c3, c4;
float d1, d2, d3;

// variables for Least-square identification
int lsid_active = 0;
float PRBSVal = 150;  // PRBS value in PWM unit
int feedin;
byte bvec[13]={0,1,1,0,0,0,1,1,0,0,1,0,1};  // unit delay output
int bj;    // index for bvec

// variables for auto-tuning function

int at_active = 0;  // autotuning active
int at_finish = 0;  // autotuning finish
int at_success = 0;  // autotuning success
int at_datacapt = 0;    // starts data capture
int at_cyclecnt=0;    // count oscillating cycle
int at_cyclestart=20;  // start measuring and computing Ku, Tu
float at_yold=0;     // previous output
float at_p=0;        // magnitude of P
float at_w=0;        // frequency of P
float at_ku=0;       //ultimate gain
float at_tu=0;       //oscillation period
float at_d=RELAYOUT;        // relay amplitude
float at_amax=0;     // min, max, center of output amplitude
float at_amin=0;
float at_acenter=0;
float at_as[4]={0,0,0,0};    // output amplitude (4 steady cycles)
float at_tus[4]={0,0,0,0};   // oscillation period (4 steady cycles)
float at_aa=0;       // output amplitude (average)
float at_tua=0;      // oscillation period (average)
float at_i=0;        // a counter to keep number of iterations
                    // between two zero crossing
float at_time=0;     // keep track of time for no oscillation
int at_dataidx=0;     // data array index
int at_datasamp=1;    // data sampling of autotune mode
int at_datacnt=0;
```

```
int adma=0;            // adma flag
int ADVal[4]={0,0,0,0};  // keep 4 samples for averaging
int ADVala;            // average of A/D value

float ep2 = 0, ep1 = 0, ep0 = 0;              // proportional-term error
float e1 = 0, e0 = 0;              // true error
float eus1 =0, eus0 = 0;              // u_sat error
float ed2 = 0, ed1 = 0, ed0 = 0;    // derivative-term error
float u2 = 0, u1 = 0, u0 = 0, u0lim = 0;     // previous and current
                                              // controller outputs
int u0limint = 0;      // integer value of controller output

// encoder union variables
typedef union
{
  byte b8[4];    // 1/4 of long variable
  long li32;
} qeidata;

qeidata QEIData, QEICmd;
int qei_index=0;

// function prototypes
void cmdInt(void);    // command interpreter function
void Controller(void);   // timer interruput routine
void PID_update(void);    // update controller coefficients

void setup() {
  pinMode(PWMOut, OUTPUT);
  pinMode(DIROut, OUTPUT);
  //setPwmFrequency(9,8);    // reduce PWM frequency of pin 9
  TCCR1B = (TCCR1B & 0b11111000) | 0x04; // reduced PWM frequency of pin 9
                                          // to 122.55 Hz
  Serial.begin(115200);    // start serial communcitaion
  Wire.begin();            // start I2C communication
  T_ms = 1000*T;
```

```
  MsTimer2::set(T_ms, Controller);    // set timer2 interrupt function
  MsTimer2::start();
  AD2V = VMAX/ADCMAX;   // compute ADC to Voltage conversion factor
  V2PWM = PWMMAX/VMAX;  // compute Voltage to PWM conversion factor
  if (inputselect==ENCIN) outscale = 1;      // default of output scaling
  if (inputselect==ADCIN) outscale = V2PWM;
  PID_update();    // compute controller coefficients
  if (outputselect==PWMOUT)  {
     pwmout = PWMMID;
     analogWrite(PWMOut, pwmout);  // send out PWM signal
  }
  else if (outputselect==PWMDIROUT)   {
     pwmout = 0;
     analogWrite(PWMOut, pwmout);  // send out PWM signal
  }
}

void loop() {

  while (Serial.available() > 0 )   {
        rcvdstring = Serial.readString();
        newcmd = 1;
  }
  if (newcmd)   { // execute this part only when new command received
    cmdInt();   // invoke the interpreter
    newcmd = 0;
  }
}

// command interpreter implementation
void cmdInt(void)
{
    rcvdstring.trim();  // remove leading&trailing whitespace, if any
    // find index of separator (blank character)
    sepIndex = rcvdstring.indexOf(' ');
    if (sepIndex==-1) {
      cmdstring = rcvdstring;
```

```
    noparm = 1;
  }
  else  {
  // extract command and parameter
    cmdstring = rcvdstring.substring(0, sepIndex);
    parmstring = rcvdstring.substring(sepIndex+1);
    noparm = 0;
  }
  // check if received command string is a valid command
  if (cmdstring.equalsIgnoreCase("step"))   {
    if (noparm==0)  {  // step to new spedified value
       parmvalfloat = parmstring.toFloat();
       if (inputselect == ADCIN)   {
        //limit step command to +/- 2.5 volts
        if (parmvalfloat > VMID) parmvalfloat = VMID;
        else if (parmvalfloat< -VMID) parmvalfloat = -VMID;
       }
       else if (inputselect == ENCIN)   {
        //limit step command to +/- 100 rounds
        if (parmvalfloat > 36000) parmvalfloat = 36000;
        else if (parmvalfloat< -36000) parmvalfloat = -36000;
       }
      cmd = parmvalfloat;
      cmdold = cmd;   // save previous command
    }
    Serial.println("datamat = [");
    datacapt = 1;    // set the flag to capture data
    i = 0;              // reset data index
  }
  else if (cmdstring.equalsIgnoreCase("datasize"))   {
    if (noparm==1)   {
      Serial.print("Current datasize = ");
      Serial.println(datasize);
    }
    else   {
      parmvalint = parmstring.toInt();
      if (parmvalint > DATASIZEMAX) parmvalint = DATASIZEMAX; // limit
```

```
                              // datasize to DATASIZEMAX
      else if (parmvalint<0) parmvalint = 0;
      datasize = parmvalint;
      if (verboseflag) {
        Serial.print("Datasize set to ");
       Serial.println(datasize);
      }
    }
  }
  // set PWM and dir output
   else if (cmdstring.equalsIgnoreCase("pwm"))   {
    if (noparm==1)   {
      Serial.print("Current pwmout = ");
      if (outputselect==PWMOUT)   {
        int ppwm = 0.7874*(pwmout - PWMMID);
        Serial.print(ppwm);
      }
      else if (outputselect==PWMDIROUT)   {
        if (dir == 1) Serial.print("-");
        int ppwm = 0.3921569*pwmout;
        Serial.print(ppwm);
      }
      Serial.println(" %");
     }
    else   {
      parmvalint = parmstring.toInt();
      if (parmvalint > 100) parmvalint = 100; // limit pwm to 100%
      else if (parmvalint<-100) parmvalint = -100;
      if (outputselect==PWMOUT)   {
        pwmout = 1.27*parmvalint+PWMMID;
      }
      else if (outputselect==PWMDIROUT)   {
        if (parmvalint>=0)   {
          pwmout = 2.55*parmvalint;
          dir = 0;
          digitalWrite(DIROut, dir^dirsense);
        }
```

```
        else   {
          parmvalint = -parmvalint;
          pwmout = 2.55*parmvalint;
          dir = 1;
          digitalWrite(DIROut, dir^dirsense);
        }
      }
      analogWrite(PWMOut,pwmout);
      if (verboseflag) {
        Serial.print("pwmout set to ");
        Serial.println(pwmout);
        if (outputselect == PWMDIROUT)   {
          Serial.print("dir set to ");
          Serial.println(dir);
        }
      }
    }
  }

  // set dirsense
  else if (cmdstring.equalsIgnoreCase("dirsense"))   {
    if (noparm==1)   {
      Serial.print("Current dirsense = ");
      Serial.println(dirsense);
    }
    else   {
      parmvalint = parmstring.toInt();
      if (parmvalint > 1) parmvalint = 1; // limit dirsense to 0,1
      else if (parmvalint<0) parmvalint = 0;
      dirsense = parmvalint;
      if (verboseflag) {
        Serial.print("dirsense set to ");
       Serial.println(dirsense);
      }
    }
  }
   else if (cmdstring.equalsIgnoreCase("verbose"))   {
```

```
    if (noparm == 1)   {
      if (verboseflag) Serial.print("Current verbose = ");
      if (verboseflag==0) Serial.println("OFF");
      else Serial.println("ON");
    }
    else   {
      if (parmstring.equalsIgnoreCase("off"))  {
        verboseflag = 0;
        if (verboseflag) Serial.println("Verbose set to OFF");
      }
      else if (parmstring.equalsIgnoreCase("on"))  {
        verboseflag = 1;
        if (verboseflag) Serial.println("Verbose set to ON");
      }
    }
}

else if (cmdstring.equalsIgnoreCase("adma"))   {
    if (noparm == 1)   {
      if (verboseflag) Serial.print("Current ADMA = ");
      if (adma==0) Serial.println("OFF");
      else Serial.println("ON");
    }
    else   {
      if (parmstring.equalsIgnoreCase("off"))  {
        adma = 0;
        if (verboseflag) Serial.println("ADMA set to OFF");
      }
      else if (parmstring.equalsIgnoreCase("on"))  {
        adma = 1;
        if (verboseflag) Serial.println("ADMA set to ON");
      }
    }
}
 else if (cmdstring.equalsIgnoreCase("insel"))   {
    if (noparm == 1)   {
      Serial.print("Input Select = ");
```

```
        if (inputselect==ADCIN) Serial.println("ADC");
        else if (inputselect==ENCIN) Serial.println("ENC");
      }
      else   {
        if (parmstring.equalsIgnoreCase("adc"))  {
          inputselect = ADCIN;
          outscale = V2PWM;
          if (verboseflag) Serial.println("input set to ADC");
        }
        else if (parmstring.equalsIgnoreCase("enc"))  {
          inputselect = ENCIN;
          outscale = 1;
          if (verboseflag) Serial.println("input set to ENC");
        }
      }
  }
  else if (cmdstring.equalsIgnoreCase("outsel"))   {
      if (noparm == 1)   {
        Serial.print("Output Select = ");
        if (outputselect==PWMOUT) Serial.println("PWM");
        else if (outputselect==PWMDIROUT) Serial.println("PWM/DIR");
      }
      else   {
        if (parmstring.equalsIgnoreCase("pwm"))  {
          outputselect = PWMOUT;
          if (verboseflag) Serial.println("output set to PWM");
        }
        else if (parmstring.equalsIgnoreCase("pwmdir"))  {
          outputselect = PWMDIROUT;
          if (verboseflag) Serial.println("input set to PWM/DIR");
        }
      }
  }
   else if (cmdstring.equalsIgnoreCase("enc"))   {
    if (noparm == 1)   {
      if (verboseflag) Serial.print("Current enc = ");
      QEIData.li32 = 0;
```

```
      qei_index = 0;
      Wire.requestFrom(0x30, 5);   // request 4 bytes from slave device
                                   // 0x30
      while (Wire.available())  {  // slave may send less than requested
        QEIData.b8[qei_index++] = Wire.read();    // read a byte
      }
      Serial.println(QEIData.li32);
    }
    else   {
      parmvalint = parmstring.toInt();

      QEICmd.li32 = parmvalint;
      if (verboseflag)   {
        // echo value to console
        Serial.print("encoder set to ");
        Serial.println(QEICmd.li32);
      }
      Wire.beginTransmission(0x30); // transmit to PIC24EP
      for (qei_index=0;qei_index<4;qei_index++)  {// send 4 bytes of data
        Wire.write(QEICmd.b8[qei_index]);
      }
      Wire.endTransmission();      // stop transmitting
    }
  }
  // switch controller type
  else if (cmdstring.equalsIgnoreCase("controller"))   {
      if (noparm == 1)   {
        if (verboseflag) Serial.print("Current controller type = ");
        if (controltype==PID) Serial.println("PID");
        else Serial.println("CC");
      }
      else   {
        if (parmstring.equalsIgnoreCase("PID"))  {
          controltype = PID;
          if (verboseflag) Serial.println("Controller set to PID");
        }
        else if (parmstring.equalsIgnoreCase("CC"))  {
```

```
          controltype = CC;
          if (verboseflag) Serial.println("Controller set to CC");
        }
      }
  }
  else if (cmdstring.equalsIgnoreCase("kp"))   {
    if (noparm == 1)   {
      if (verboseflag) Serial.print("Current kp = ");
      Serial.println(kp);
    }
    else   {
      parmvalfloat = parmstring.toFloat();
      if (parmvalfloat > KPMAX) parmvalfloat = KPMAX; // limit kp value
      else if (parmvalfloat<0) parmvalfloat = 0;
      kp = parmvalfloat;
      if (verboseflag)   {
        // echo value to console
        Serial.print("new kp = ");
        Serial.println(kp);
      }
      PID_update();    // update controller coefficients
    }
  }
  else if (cmdstring.equalsIgnoreCase("ki"))   {
    if (noparm == 1)   {
      if (verboseflag) Serial.print("Current ki = ");
      Serial.println(ki);
    }
    else   {
      parmvalfloat = parmstring.toFloat();
      if (parmvalfloat > KIMAX) parmvalfloat = KIMAX; // limit ki value
      else if (parmvalfloat<0) parmvalfloat = 0;
      ki = parmvalfloat;
      if (verboseflag)   {
        // echo value to console
        Serial.print("new ki = ");
        Serial.println(ki);
```

```
      }
      PID_update();    // update controller coefficients
    }
  }
  else if (cmdstring.equalsIgnoreCase("kd"))   {
    if (noparm == 1)   {
      if (verboseflag) Serial.print("Current kd = ");
      Serial.println(kd);
    }
    else   {
      parmvalfloat = parmstring.toFloat();
      if (parmvalfloat > KDMAX) parmvalfloat = KDMAX; // limit kd value
      else if (parmvalfloat<0) parmvalfloat = 0;
      kd = parmvalfloat;
      if (verboseflag)   {
        // echo value to console
        Serial.print("new kd = ");
        Serial.println(kd);
      }
      PID_update();    // update controller coefficients
    }
   }
  else if (cmdstring.equalsIgnoreCase("kt"))   {
    if (noparm == 1)   {
      if (verboseflag) Serial.print("Current kt = ");
      Serial.println(kt);
    }
    else   {
      parmvalfloat = parmstring.toFloat();
      if (parmvalfloat > KTMAX) parmvalfloat = KTMAX; // limit kt value
      else if (parmvalfloat<0) parmvalfloat = 0;
      kt = parmvalfloat;
      if (verboseflag)   {
        // echo value to console
        Serial.print("new kt = ");
        Serial.println(kt);
      }
```

```
    PID_update();    // update controller coefficients
  }
 }
   else if (cmdstring.equalsIgnoreCase("wp"))   {
  if (noparm == 1)   {
    if (verboseflag) Serial.print("Current wp = ");
    Serial.println(wp);
  }
  else   {
    parmvalfloat = parmstring.toFloat();
    if (parmvalfloat > WPMAX) parmvalfloat = WPMAX; // limit wp value
    else if (parmvalfloat<0) parmvalfloat = 0;
    wp = parmvalfloat;
    if (verboseflag)   {
      // echo value to console
      Serial.print("new wp = ");
      Serial.println(wp);
    }
    PID_update();    // update controller coefficients
  }
 }
 else if (cmdstring.equalsIgnoreCase("wd"))   {
  if (noparm == 1)   {
    if (verboseflag) Serial.print("Current wd = ");
    Serial.println(wd);
  }
  else   {
    parmvalfloat = parmstring.toFloat();
    if (parmvalfloat > WDMAX) parmvalfloat = WDMAX; // limit wd value
    else if (parmvalfloat<0) parmvalfloat = 0;
    wd = parmvalfloat;
    if (verboseflag)   {
      // echo value to console
      Serial.print("new wd = ");
      Serial.println(wd);
    }
    PID_update();    // update controller coefficients
```

```
 }
}
else if (cmdstring.equalsIgnoreCase("autotune"))   {  // start PID
                                                      // autotuning
 at_acenter = y;
 at_amax = y;
 at_amin = y;
 at_cyclecnt = 0;    // start counting oscillating cycle
 at_i = 0;
 loopclose = 0;    //open feedback loop
 at_finish = 0;
 at_success = 0;
 at_datacapt = 0;
 at_active = 1;
 if (verboseflag) Serial.println("PID autotuning starts");
 // debug
 Serial.print("at_acenter = ");
 Serial.println(at_acenter);
}
 else if (cmdstring.equalsIgnoreCase("lsid"))   {  // start LS
                                                  // identification
  loopclose = 0;    // open feedback loop
  if (verboseflag) Serial.println("LSID starts");
  lsid_active = 1;
   Serial.println("datamat = [");
   datacapt = 1;    // set the flag to capture data
   i = 0;              // reset data index
}
else if (cmdstring.equalsIgnoreCase("open"))   {  // open feedback
                                                  // loop
  loopclose = 0;
  if (verboseflag) Serial.println("feedback loop open");
}
else if (cmdstring.equalsIgnoreCase("close"))   {  // close feedback
                                                   // loop
  loopclose = 1;
  if (verboseflag) Serial.println("feedback loop closed");
}
else if (cmdstring.equalsIgnoreCase("fbloop"))   {  // check loop
```

```
                                                          // status
      if (loopclose == 0)
           Serial.println("open");
      else  Serial.println("close");
    }
   else if (cmdstring.equalsIgnoreCase("y"))   {  // print plant output
       if (verboseflag) Serial.print("y = ");
       Serial.println(y);
   }
   else   {
     Serial.println("Invalid command");
   }
}

// update controller coefficients
void PID_update(void)
{
  double x1, x2;
  x1 = 1 + N*T;
  x2 = 2 + N*T;
  a1 = x2/x1;
  a2 = -1/x1;
  b1 = kp;
  b2 = -kp*x2/x1;
  b3 = kp/x1;
  c1 = ki*T;
  c2 = -ki*T/x1;
  c3 = kt*T;
  c4 = -kt*T/x1;
  d1 = kd*N/x1;
  d2 = -2*kd*N/x1;
  d3 = kd*N/x1;
}

// timer interrupt routine
void Controller(void)
{
```

```
sei();    // enable interrupt
// update previous values
ep2 = ep1;  // proportional term error
ep1 = ep0;
e1 = e0;    // true error
eus1 = eus0;    // u_sat error
ed2 = ed1;    // derivative term error
ed1 = ed0;
u2 = u1;
u1 = u0;
if (inputselect == ADCIN)   {
  // A/D averaging
  ADVal[3]=ADVal[2];
  ADVal[2]=ADVal[1];
  ADVal[1]=ADVal[0];
  ADVal[0]=analogRead(VO);
  if (adma)
    ADVala = (ADVal[0]+ADVal[1]+ADVal[2]+ADVal[3])>>2;  // average of 4
                                                        // samples
  else ADVala = ADVal[0];
  y = AD2V*(ADVala - ADCMID); // read output voltage and apply offset
                              // y has unit of volts
}
else if (inputselect == ENCIN)   {
  qei_index = 0;
  Wire.requestFrom(0x30, 5);   // request 4 bytes from slave device 0x30
  while (Wire.available())  {  // slave may send less than requested
    QEIData.b8[qei_index++] = Wire.read();    // read a byte
  }
  y = 0.075*QEIData.li32;  // y has unit of degrees
}
e0 = cmd - y;        // compute true error
ep0 = wp*cmd - y;    // proportional-term error
ed0 = wd*cmd - y;    // derivative-term error
// u-sat error
if (inputselect == ADCIN)  {
  if (u0 > VMID) eus0 = VMID - u0;
```

```
    else if (u0<-VMID) eus0 = -u0 - VMID;
    else eus0 = 0;
  }
  else if (inputselect == ENCIN)   {
    if (outputselect == PWMOUT)   {
      if (u0 > PWMMID) eus0 = PWMMID - u0;
      else if (u0<-PWMMID) eus0 = -u0 - PWMMID;
      else eus0 = 0;
    }
    else if (outputselect == PWMDIROUT)   {
      if (u0 > PWMMAX) eus0 = PWMMAX - u0;
      else if (u0<-PWMMAX) eus0 = -u0 - PWMMAX;
      else eus0 = 0;
    }
  }
  if (loopclose == 0)   {    // loop open
    if (at_active)   {    // in autotuning mode
      if (at_finish == 0)   {  // autotune has not finished
        if (y <at_acenter)     // drive in + direction
          u0lim = at_d*outscale;      // relay amplitude (in PWM unit)
        else                   // drive in - direction
          u0lim = -at_d*outscale;
       if (outputselect==PWMOUT)   {  // output PWM only
        if (u0lim > PWMMID) u0lim = PWMMID;
        else if (u0lim < -PWMMID) u0lim = -PWMMID;
        u0limint = int(u0lim);
        pwmout = u0limint + PWMMID;
      }
      else if (outputselect==PWMDIROUT)   {  // output PWM and direction
        if (u0lim>=0) {    // positive controller output
           dir = 0;
        }
        else   {
          u0lim = -u0lim;    // invert polarity
          dir = 1;
        }
        if (u0lim > PWMMAX) u0lim = PWMMAX;
```

```
        else if (u0lim < 0) u0lim = 0;
        pwmout = int(u0lim);
        digitalWrite(DIROut, dir^dirsense);  // direction can be switched
                                              // by dirsense
      }
      analogWrite(PWMOut, pwmout);  // send out PWM signal
        // detect magnitude and period of oscillation
        if (y > at_amax) at_amax = y;
        if (y < at_amin) at_amin = y;  // amin is not used to compute Ku
        if ((at_yold<at_acenter)&&(y>=at_acenter))   { // center crossing
          if (at_datacapt==0) Serial.println(at_cyclecnt);
          at_tu = at_i*T;
          at_cyclecnt++;
          at_i = 0;
          if (at_cyclecnt >= at_cyclestart)   { // start processing data
            if ((at_datacapt==0)&&(verboseflag))  {
              at_datacapt = 1;    // also capture data if verbose ON
            }
            at_dataidx = at_cyclecnt - at_cyclestart;
            at_as[at_dataidx] = at_amax - at_acenter;
            at_tus[at_dataidx] = at_tu;
          }
          if (at_dataidx == 3)    // average for 4 cycles
          {
            at_dataidx = 0;
            at_datacapt = 0;
            at_finish = 1;
          }
        }  // if ((at_yold<at_acenter)&&(y>=at_acenter))
        else   {  // quit if no oscillation after MAXATTIME passeed
          at_time = at_i*T;
          if (at_time > MAXATTIME)  {
            at_active = 0;    // quit autotuning mode
            if (verboseflag) Serial.println("No oscillation deteceted. PID
Autotuning fails!");
          }
        }  // else clause of if ((at_yold<at_acenter)&&(y>=at_acenter))
        if (at_datacapt)   {    // send data to host
```

```
      at_datacnt++;
      if (at_datacnt == at_datasamp)   {
        if (outputselect==PWMDIROUT)
          if (dir==1) Serial.print("-");
        Serial.print(u0lim);
        Serial.print(", ");
        Serial.print(y);
        Serial.println(";");
        at_datacnt = 0;
       }
     }
    at_yold = y;
    at_i++;
  }   // if (at_finish == 0)
  else  {  // compute PID parameters
    at_aa = (at_as[0]+at_as[1]+at_as[2]+at_as[3])/4;
    at_tu = (at_tus[0]+at_tus[1]+at_tus[2]+at_tus[3])/4;
    at_ku = (28*at_d)/(22*at_aa);
    kp = 0.6*at_ku;
    ki = 1.2*at_ku/at_tu;
    kd = 0.075*at_ku*at_tu;
    PID_update();      // update controller coefficients
    if (verboseflag)   {
      Serial.print("Ku = ");
      Serial.println(at_ku);
      Serial.print("Tu = ");
      Serial.println(at_tu);
      Serial.print("Kp = ");
      Serial.println(kp);
      Serial.print("Ki = ");
      Serial.println(ki);
      Serial.print("Kd = ");
      Serial.println(kd);
    }
    at_success = 1;
    at_active = 0;
    // set pwm output to 0
```

```
      if (outputselect == PWMDIROUT)   {
        pwmout = 0;
        analogWrite(PWMOut, pwmout);  // send out PWM signal
      }
      if (verboseflag) Serial.println("Autotuning ends.");
    }
  }      // if (at_active)
  else if (lsid_active)  { // excite the plant with PRBS input
    // generate PRBS input
    feedin = bvec[0]^(bvec[2]^(bvec[3]^bvec[12]));
    for (bj = 12;bj>=1;bj--) {
         bvec[bj]=bvec[bj-1];
    }
    bvec[0] = feedin;
    if (bvec[12]==0) u0 = -PRBSVal;
    else u0 = PRBSVal;
    u0lim = u0*outscale;
    if (outputselect==PWMOUT)   {  // output PWM only
      if (u0lim > PWMMID) u0lim = PWMMID;
      else if (u0lim < -PWMMID) u0lim = -PWMMID;
      u0limint = int(u0lim);
      pwmout = u0limint + PWMMID;
    }
    else if (outputselect==PWMDIROUT)   {  // output PWM and direction
      if (u0lim>=0) {    // positive controller output
         dir = 0;
      }
      else   {
        u0lim = -u0lim;    // invert polarity
        dir = 1;
      }
      if (u0lim > PWMMAX) u0lim = PWMMAX;
      else if (u0lim < 0) u0lim = 0;
      pwmout = int(u0lim);
      if (dir==1) u0limint = -pwmout;
      else u0limint = pwmout;
      digitalWrite(DIROut, dir^dirsense);  // direction can be switched
```

```
                                                // by dirsense
      }
      analogWrite(PWMOut, pwmout);  // send out PWM signal
    }   // if (lsid_active)
    else   {
    // uncomment this part to implement open-loop control
    // It cannot be applied to DC motor position control
//      u0 = cmd;
//      u0lim = outscale*u0;    // scale output if necessary
//
//      if (outputselect==PWMOUT)   {  // output PWM only
//          if (u0lim > PWMMID) u0lim = PWMMID;
//          else if (u0lim < -PWMMID) u0lim = -PWMMID;
//          u0limint = int(u0lim);
//          pwmout = u0limint + PWMMID;
//      }
//      else if (outputselect==PWMDIROUT)   {  // output PWM and direction
//          if (u0lim>=0) {    // positive controller output
//             dir = 0;
//          }
//          else   {
//            u0lim = -u0lim;    // invert polarity
//            dir = 1;
//          }
//
//          if (u0lim > PWMMAX) u0lim = PWMMAX;
//          else if (u0lim < 0) u0lim = 0;
//          pwmout = int(u0lim);
//          if (dir==1) u0limint = -pwmout;
//          else u0limint = pwmout;
//          digitalWrite(DIROut, dir^dirsense);  // direction can be
                                                 // switched by dirsense
//      }  // if (outputselect==)
//
//      analogWrite(PWMOut, pwmout);  // send out PWM signal
```

```
    }   // else clause of if(at_active)


  } // if (loopclose==0)
  else   {        // loop close
    if (controltype==PID)   {
      u0   =   a1*u1+a2*u2+b1*ep0+b2*ep1+b3*ep2+c1*e0+c2*e1+c3*eus0+c3*eus0
+c4*eus1+d1*ed0+d2*ed1+d3*ed2;
    }
    else if (controltype == CC)   {
        // update states
        for (ix=ccnumber-1;ix>0;ix--)
          x[ix] = x[ix-1];
        x[0] = e0;
        for (ix = 1; ix<ccnumber; ix++)
          x[0]-=a[ix]*x[ix];
        // x[0] /= a[0];  // in case a[0] != 1
        u0 = 0;
        for (ix=0; ix<ccnumber; ix++)
          u0+=b[ix]*x[ix];
    }
    u0lim = outscale*u0;
    if (outputselect==PWMOUT)   {  // output PWM only
      if (u0lim > PWMMID) u0lim = PWMMID;
      else if (u0lim < -PWMMID) u0lim = -PWMMID;
      u0limint = int(u0lim);
      pwmout = u0limint + PWMMID;
    }
    else if (outputselect==PWMDIROUT)   {  // output PWM and direction
      if (u0lim>=0) {    // positive controller output
         dir = 0;
      }
      else   {
        u0lim = -u0lim;    // invert polarity
        dir = 1;
      }
      if (u0lim > PWMMAX) u0lim = PWMMAX;
      else if (u0lim < 0) u0lim = 0;
      pwmout = int(u0lim);
```

```
      if (dir==1) u0limint = -pwmout;
      else u0limint = pwmout;
      digitalWrite(DIROut, dir^dirsense);  // direction can be switched by
                                                 // dirsense
    }
    analogWrite(PWMOut, pwmout);  // send out PWM signal
  } // else clause of if (loopclose == 0)
  if (datacapt)   {
    // send data to host
    Serial.print(cmd);
    Serial.print(", ");
    Serial.print(y);
    Serial.print(", ");
    Serial.print(u0);
    Serial.print(", ");
    Serial.print(u0limint);
    Serial.println(";");
    i++;
    if (i== datasize)  {
      datacapt = 0;      // reset the command change flag
      i = 0;              // reset index
      Serial.println("];");
      if (lsid_active) lsid_active = 0;    // quit LSID
    }
  }
}
```

Listing B.1 `FX.ino` complete FX sketch for Arduino UNO