```
/*****************************************************************************
 **
 **  LVFA_Firmware - Provides Basic Arduino Sketch For Interfacing With
 **  LabVIw
**
 **  Written By:    Sam Kristoff - National Instruments
 **  Written On: November 2010
 **  Last Updated:  Dec 2011 - Kevin Fort - National Instruments
 **
 **  This File May Be Modified And Re-Distributed Freely.
 **  Written By Sam Kristoff And Available At www.ni.com/arduino.
 **
*****************************************************************************/


/*****************************************************************************
 **
 ** Includes.
 **
*****************************************************************************/
// Standard includes. These should always be
included. #include <Wire.h>
#include <SPI.h>
#include <Servo.h>
#include "LabVIEWInterface.h"
#include <Servo.h>

Servo  myservo;

const int  buttonPin    =   2;// the number of the pushbutton pin


// variables will change:
int buttonState = 0;

/*****************************************************************************
 **  setup()
**
 **  Initialize the Arduino and setup serial communication.
**
 **  Input:  None
 **  Output: None
*****************************************************************************/
void setup()
{
  // Initialize Serial Port With The Default Baud Rate
  syncLV();
```

```
    // initialize the
    pushbutton pin as an
    input:
    pinMode(buttonPin,
    INPUT);

     myservo.attach(10);
  }


  /*****************************************************
  ******************
   **  loop()
**
   **  The main loop.  This loop runs continuously on the
       Arduino.  It
   **  receives and processes serial commands from
       LabVIEW.
   **
   **  Input:  None
   **  Output: None
*****************************************************
******************
 void loop()
 {
    // Check for commands from LabVIEW and process them.

   checkForCommand();
    // Place your custom loop code here (this may slow
      down communication with L buttonState
      =digitalRead(buttonPin);

    // check if the pushbutton is pressed.
    // if it is, the buttonState is HIGH:
    if (buttonState  ==HIGH)      {
      //  turn  LED  on:
      myservo.write(100);
    else {                              }
      // turn LED off:
      myservo.write(0);
    }

    if(acqMode==1)
    {
      sampleContinously();
```

```
Adafruit Motor shield library
// copyright Adafruit Industries LLC, 2009
// this code is public domain, enjoy!

#ifndef _AFMotor_h_
#define _AFMotor_h_

#include <inttypes.h>
#include <avr/io.h>

//#define MOTORDEBUG 1

#define     MICROSTEPS     16// 8 or 16

#define MOTOR12_64KHZ _BV(CS20)// no prescale
#define MOTOR12_8KHZ _BV(CS21)// divide by 8
#define MOTOR12_2KHZ _BV(CS21) | _BV(CS20)// divide by
32 #define MOTOR12_1KHZ _BV(CS22)// divide by 64

#define MOTOR34_64KHZ _BV(CS00)// no prescale
#define MOTOR34_8KHZ _BV(CS01)// divide by 8
#define MOTOR34_1KHZ _BV(CS01) | _BV(CS00)// divide by 64

#define MOTOR1_A 2
#define MOTOR1_B 3
#define MOTOR2_A 1
#define MOTOR2_B 4
#define MOTOR4_A 0
#define MOTOR4_B 6
#define MOTOR3_A 5
#define MOTOR3_B 7

#define FORWARD 1
#define BACKWARD
2 #define BRAKE 3
#define RELEASE 4

#define SINGLE 1
#define DOUBLE 2
#define INTERLEAVE 3
#define MICROSTEP 4

/*
#define LATCH 4
#define LATCH_DDR DDRB
```

```
#define LATCH_PORT PORTB

#define CLK_PORT
PORTD #define CLK_DDR
DDRD #define CLK 4

#define ENABLE_PORT
PORTD #define ENABLE_DDR
DDRD #define ENABLE 7

#define SER 0
#define SER_DDR DDRB
#define SER_PORT
PORTB */

// Arduino pin names
#define MOTORLATCH 12
#define MOTORCLK 4
#define MOTORENABLE 7
#define MOTORDATA 8

class AFMotorController
{
  public:
    AFMotorController(void);
    void enable(void);
      friendclass AF_DCMotor;
     void latch_tx(void);
};

class AF_DCMotor
{
 public:
  AF_DCMotor(uint8_t motornum, uint8_t freq =
  MOTOR34_8KHZ); void run(uint8_t);
  void setSpeed(uint8_t);

 private:
  uint8_t motornum,
pwmfreq; };

class AF_Stepper
 { public:
 AF_Stepper(uint16_t, uint8_t);
  void step(uint16_t steps, uint8_t dir,  uint8_t style = SINGLE);
```

```
  void setSpeed(uint16_t);
  uint8_t onestep(uint8_t dir, uint8_t
  style); void release(void);
  uint16_t revsteps;// # steps per
  revolution uint8_t steppernum;
  uint32_t usperstep,
 steppingcounter; private:
  uint8_t currentstep;

};

uint8_t getlatchstate(void);

#endif
```

```
// AccelStepper.cpp
//
// Copyright (C) 2009 Mike McCauley
// $Id: AccelStepper.cpp,v 1.4 2011/01/05 01:51:01 mikem Exp $

 #if defined(ARDUINO) && ARDUINO >=
   100 #include"Arduino.h"
  #else
  #include"WProgram.h"
  #endif

#include "AccelStepper.h"

void AccelStepper::moveTo(long absolute)
{
    _targetPos = absolute;
    computeNewSpeed();
}

void AccelStepper::move(long relative)
{
    moveTo(_currentPos + relative);
}

// Implements steps according to the current speed
// You must call this at least once per step
// returns true if a step occurred
boolean AccelStepper::runSpeed()
{
    unsigned long time  =micros();

//    if ( (time >= (_lastStepTime + _stepInterval))  // okay if both
//         || ((time < _lastRunTime) && (time > (0xFFFFFFFF-(_lastStepTim

//    unsigned long nextStepTime = _lastStepTime + _stepInterval;
//    if (   ((nextStepTime < _lastStepTime) && (time < _lastStepTime) &&
//        || ((nextStepTime >= _lastStepTime) && (time >= nextStepTime)))

// TESTING:
//time += (0xffffffff - 10000000);


    // Gymnastics to detect wrapping of either the nextStepTime and/or
    th unsigned long nextStepTime = _lastStepTime + _stepInterval;
    if (   ((nextStepTime >= _lastStepTime) && ((time >= nextStepTime) ||
```

```
        || ((nextStepTime < _lastStepTime) && ((time >= nextStepTime) &&

    {
        if (_speed > 0.0f)
        {
            // Clockwise
            _currentPos += 1;
        }
        else if (_speed < 0.0f)
        {
            // Anticlockwise
            _currentPos -= 1;
        }
step(_currentPos & 0x7); // Bottom 3 bits (same as mod 8, but wo

//        _lastRunTime = time;
        _lastStepTime = time;
        return true;
    }
    else
    {
//        _lastRunTime =
        time; return false;
    }
}

long AccelStepper::distanceToGo()
{
    return _targetPos - _currentPos;
}

long AccelStepper::targetPosition()
{
    return _targetPos;
}

longAccelStepper::currentPosition()
{
    return _currentPos;
}

// Useful during initialisations or after initial positioning
voidAccelStepper::setCurrentPosition(long position)
{
    _targetPos = _currentPos =position;
```

```
    computeNewSpeed();// Expect speed of 0
}

voidAccelStepper::computeNewSpeed()
{
    setSpeed(desiredSpeed());
}

// Work out and return a new speed.
// Subclasses can override if they want
// Implement acceleration, deceleration and max speed
// Negative speed is anticlockwise
// This is called:
//  after each step
//  after user changes:
//   maxSpeed
//   acceleration
//   target position (relative or absolute)
float AccelStepper::desiredSpeed()
{
    float requiredSpeed;
    long distanceTo = distanceToGo();

    // Max possible speed that can still decelerate in the available dist
    // Use speed squared to avoid using sqrt
    if (distanceTo == 0)
        return  0.0f;// We're there
    else if (distanceTo > 0)// Clockwise
        requiredSpeed = (2.0f * distanceTo *
    _acceleration); else // Anticlockwise
        requiredSpeed = -(2.0f * -distanceTo * _acceleration);

    float sqrSpeed = (_speed * _speed) * ((_speed > 0.0f) ? 1.0f : -
    1.0f) if (requiredSpeed > sqrSpeed)
    {
        if (_speed  ==  _maxSpeed)// Reduces processor load by avoiding
        {
            // Maintain max speed
            requiredSpeed = _maxSpeed;
        }
        else
        {
            // Need to accelerate in clockwise
            direction if (_speed == 0.0f)
                requiredSpeed =sqrt(2.0f * _acceleration);
```

```
            else
                requiredSpeed = _speed +abs(_acceleration /
            _speed); if (requiredSpeed > _maxSpeed)
                requiredSpeed = _maxSpeed;
        }
    }
    else if (requiredSpeed < sqrSpeed)
    {
        if (_speed  ==  -_maxSpeed)// Reduces processor load by avoiding
        {
            // Maintain max speed
            requiredSpeed = -_maxSpeed;
        }
        else
        {
            // Need to accelerate in clockwise
            direction if (_speed == 0.0f)
                requiredSpeed = -sqrt(2.0f *
            _acceleration); else
                requiredSpeed = _speed -abs(_acceleration /
            _speed); if (requiredSpeed < -_maxSpeed)
                requiredSpeed = -_maxSpeed;
        }
    }
    else // if (requiredSpeed ==
        sqrSpeed) requiredSpeed = _speed;

// Serial.println(requiredSpeed);
    return requiredSpeed;
}

// Run the motor to implement speed and acceleration in order to proceed
// You must call this at least once per step, preferably in your main loo
// If the motor is in the desired position, the cost is very small
// returns true if we are still running to position
boolean AccelStepper::run()
{
    if (_targetPos == _currentPos)
        return false;

    if (runSpeed())
        computeNewSpeed();
    return true;
}
```

```
AccelStepper::AccelStepper(uint8_t pins, uint8_t pin1, uint8_t pin2, uint
{
    _pins = pins;
    _currentPos = 0;
    _targetPos = 0;
    _speed = 0.0;
    _maxSpeed = 1.0;
    _acceleration = 1.0;
    _stepInterval = 0;
// _lastRunTime = 0;
    _minPulseWidth =
    1; _lastStepTime =
    0; _pin1 = pin1;
    _pin2 = pin2;
    _pin3 = pin3;
    _pin4 = pin4;
//_stepInterval = 20000;
//_speed = 50.0;
//_lastRunTime = 0xffffffff - 20000;
//_lastStepTime = 0xffffffff - 20000 - 10000;
    enableOutputs();
}

AccelStepper::AccelStepper(void  (*forward)(),void (*backward)())
{
    _pins = 0;
    _currentPos = 0;
    _targetPos = 0;
    _speed = 0.0;
    _maxSpeed = 1.0;
    _acceleration = 1.0;
    _stepInterval = 0;
// _lastRunTime = 0;
    _minPulseWidth =
    1; _lastStepTime =
    0; _pin1 = 0;
    _pin2 = 0;
    _pin3 = 0;
    _pin4 = 0;
    _forward = forward;
    _backward = backward;
}

void AccelStepper::setMaxSpeed(float speed)
{
```

```
     _maxSpeed =speed;
    computeNewSpeed();
}

void AccelStepper::setAcceleration(float acceleration)
{
    _acceleration = acceleration;
    computeNewSpeed();
}

void AccelStepper::setSpeed(float speed)
{
    if (speed ==
        _speed) return;

    if ((speed > 0.0f) && (speed >
        _maxSpeed)) _speed = _maxSpeed;
    else if ((speed < 0.0f) && (speed < -
        _maxSpeed)) _speed = -_maxSpeed;
    else
         _speed =speed;

     _stepInterval =abs(1000000.0 / _speed);
}

float AccelStepper::speed()
{
    return _speed;
}

// Subclasses can override
void AccelStepper::step(uint8_t step)
{
    switch (_pins)
    {
        case 0:
            step0();
            break;
         case 1:
             step1(step);
             break;

         case 2:
             step2(step);
             break;
```

```
            case 4:
                step4(step);
                break;

            case 8:
                step8(step);
                break;
        }
}

// 0 pin step function (ie for functional
usage) void AccelStepper::step0()
{
  if (_speed > 0)
   { _forward();
   }else {
   _backward();
  }
}

// 1 pin step function (ie for stepper drivers)
// This is passed the current step number (0 to 7)
// Subclasses can override
void AccelStepper::step1(uint8_tstep)
{
    digitalWrite(_pin2, _speed > 0);// Direction
    // Caution 200ns setup time
    digitalWrite(_pin1, HIGH);
    // Delay the minimum allowed pulse width
    delayMicroseconds(_minPulseWidth);
    digitalWrite(_pin1, LOW);
}

// 2 pin step function
// This is passed the current step number (0 to 7)
// Subclasses can override
void AccelStepper::step2(uint8_tstep)
{
    switch (step & 0x3)
    {
          case 0:/* 01 */
              digitalWrite(_pin1, LOW);
              digitalWrite(_pin2,
              HIGH); break;
```

```
            case 1:/* 11 */
                digitalWrite(_pin1, HIGH);
                digitalWrite(_pin2, HIGH);
                break;

            case 2:/* 10 */
                digitalWrite(_pin1, HIGH);
                digitalWrite(_pin2, LOW);
                break;

            case 3:/* 00 */
                digitalWrite(_pin1, LOW);
                digitalWrite(_pin2, LOW);
                break;
        }
}

// 4 pin step function for half stepper
// This is passed the current step number (0 to 7)
// Subclasses can override
void AccelStepper::step4(uint8_tstep)
{
    switch (step & 0x3)
    {
            case  0:// 1010
                digitalWrite(_pin1, HIGH);
                digitalWrite(_pin2, LOW);
                digitalWrite(_pin3, HIGH);
                digitalWrite(_pin4, LOW);
                break;

            case  1:// 0110
                digitalWrite(_pin1, LOW);
                digitalWrite(_pin2, HIGH);
                digitalWrite(_pin3, HIGH);
                digitalWrite(_pin4, LOW);
                break;

            case 2://0101
                digitalWrite(_pin1, LOW);
                digitalWrite(_pin2, HIGH);
                digitalWrite(_pin3, LOW);
                digitalWrite(_pin4, HIGH);
                break;
```

```
        case 3://1001
            digitalWrite(_pin1, HIGH);
            digitalWrite(_pin2, LOW);
            digitalWrite(_pin3, LOW);
            digitalWrite(_pin4, HIGH);
            break;
    }
}


// 4 pin step function
// This is passed the current step number (0 to 3)
// Subclasses can override
void AccelStepper::step8(uint8_tstep)
{
    switch (step & 0x7)
    {
        case  0:// 1000
           digitalWrite(_pin1, HIGH);
           digitalWrite(_pin2, LOW);
           digitalWrite(_pin3, LOW);
           digitalWrite(_pin4, LOW);
           break;

        case  1:// 1010
           digitalWrite(_pin1, HIGH);
           digitalWrite(_pin2, LOW);
           digitalWrite(_pin3, HIGH);
           digitalWrite(_pin4, LOW);
           break;

        case  2:// 0010
           digitalWrite(_pin1, LOW);
           digitalWrite(_pin2, LOW);
           digitalWrite(_pin3, HIGH);
           digitalWrite(_pin4, LOW);
           break;

        case  3:// 0110
           digitalWrite(_pin1, LOW);
           digitalWrite(_pin2, HIGH);
           digitalWrite(_pin3, HIGH);
           digitalWrite(_pin4, LOW);
           break;
```

```
            case  4:// 0100
                digitalWrite(_pin1, LOW);
                digitalWrite(_pin2, HIGH);
                digitalWrite(_pin3, LOW);
                digitalWrite(_pin4, LOW);
                break;

            case 5://0101
                digitalWrite(_pin1, LOW);
                digitalWrite(_pin2, HIGH);
                digitalWrite(_pin3, LOW);
                digitalWrite(_pin4, HIGH);
                break;

            case  6:// 0001
                digitalWrite(_pin1, LOW);
                digitalWrite(_pin2, LOW);
                digitalWrite(_pin3, LOW);
                digitalWrite(_pin4,
                HIGH); break;

            case 7://1001
                digitalWrite(_pin1, HIGH);
                digitalWrite(_pin2, LOW);
                digitalWrite(_pin3, LOW);
                digitalWrite(_pin4, HIGH);
                break;
        }
    }

    // Prevents power consumption on the outputs
    void AccelStepper::disableOutputs()
    {
        if (! _pins)return;

        digitalWrite(_pin1, LOW);
        digitalWrite(_pin2, LOW);
        if (_pins == 4 || _pins == 8)
        {
            digitalWrite(_pin3, LOW);
            digitalWrite(_pin4, LOW);
        }
    }
```

```
void   AccelStepper::enableOutputs()
{
    if (!  _pins)return;

    pinMode(_pin1, OUTPUT); pinMode(_pin2,
    OUTPUT);
    if (_pins == 4 || _pins == 8)
    {
         pinMode(_pin3, OUTPUT); pinMode(_pin4,
         OUTPUT);
    }
}

voidAccelStepper::setMinPulseWidth(unsigned int minWidth)
{
  _minPulseWidth = minWidth;
}

// Blocks until the target position is reached void
AccelStepper::runToPosition()
{
    while (run())
         ;
}

booleanAccelStepper::runSpeedToPosition()
{
    return _targetPos!=_currentPos ? runSpeed() :false;
}

// Blocks until the new target position is reached
voidAccelStepper::runToNewPosition(long position)
{
    moveTo(position);
    runToPosition();
```

```
/*****************************************************************************
 **
 **
 **  LVFA_Firmware - Provides Functions For Interfacing With The Arduino
**
 **  Written By:    Sam Kristoff - National Instruments
 **  Written On: November 2010
 **  Last Updated:  Dec 2011 - Kevin Fort - National Instruments
 **
 **  This File May Be Modified And Re-Distributed Freely. Original File C
 **  Written By Sam Kristoff And Available At www.ni.com/arduino.
 **
*****************************************************************************


 /*****************************************************************************
 **  Define Constants
**
 **  Define directives providing meaningful names for constant values.
 *****************************************************************************

 #define     FIRMWARE_MAJOR 02
 #define  FIRMWARE_MINOR  00
 #if defined(__AVR_ATmega1280__) || defined(__AVR_ATmega2560__)
 #define  DEFAULTBAUDRATE  115200// Defines The Default Serial Baud Rate
 #else
 #define DEFAULTBAUDRATE 115200
 #endif
 #define
 #define
 #define


 // Declare Variables
 unsigned char currentCommand[COMMANDLENGTH];// The Current Command Fo
 //Globals for continuous aquisition
 unsigned char acqMode;
 unsigned char contAcqPin;
 float contAcqSpeed; float
 acquisitionPeriod; float
 iterationsFlt;
 int iterations;
 float delayTime;
```

```
/*****************************************************************
**  syncLV
**
**  Synchronizes with LabVIEW and sends info about the board and firmware
**
**  Input:  None
**  Output: None
*****************************************************************
void syncLV();

/*****************************************************************
**  setMode
**
**  Sets the mode of the Arduino (Reserved For Future Use)
**
**  Input:  Int - Mode
**  Output: None
*****************************************************************
void setMode(int mode);

/*****************************************************************
**  checkForCommand
**
**  Checks for new commands from LabVIEW and processes them if any exists
**
**  Input:  None
**  Output: 1 - Command received and processed
**          0 - No new command
*****************************************************************
int checkForCommand(void);

/*****************************************************************
**  processCommand
**
**  Processes a given command
**
**  Input:  command of COMMANDLENGTH bytes
**  Output: 1 - Command received and processed
**          0 - No new command
*****************************************************************
void processCommand(unsigned char command[]);

/***************************************************************** **
writeDigitalPort
**
```

```
**  Write values to DIO pins 0 - 13.  Pins must first be configured as ou
**
**  Input:  Command containing digital port data
**  Output: None
*************************************************************************
void writeDigitalPort(unsigned char command[]);

/*************************************************************************
**  analogReadPort
**
**  Reads all 6 analog input ports, builds 8 byte packet, send via RS232.
**
**  Input:  None
**  Output: None
*************************************************************************
void analogReadPort();

/*************************************************************************
**  sevenSegment_Config
**
**  Configure digital I/O pins to use for seven segment display.  Pins ar
**
**  Input:  Pins to use for seven segment LED [A, B, C, D, E, F, G, DP]
**  Output: None
*************************************************************************
void sevenSegment_Config(unsigned char command[]);

/*************************************************************************
**  sevenSegment_Write
**
**  Write values to sevenSegment display.  Must first use sevenSegment_Co
**
**  Input:  Eight values to write to seven segment display
**  Output: None
*************************************************************************
void sevenSegment_Write(unsigned char command[]);

/*************************************************************************
**  spi_setClockDivider
**
**  Set the SPI Clock Divisor
**
**  Input:  SPI Clock Divider 2, 4, 8, 16, 32, 64, 128
**  Output: None
*************************************************************************
```

```
void spi_setClockDivider(unsigned char divider);

/*****************************************************************
**  spi_sendReceive
**
**  Sens / Receive SPI Data
**
**  Input:  Command Packet
**  Output: None (This command sends one serail byte back to LV for each
*****************************************************************
void spi_sendReceive(unsigned char command[]);

/*****************************************************************
**  checksum_Compute
**
**  Compute Packet Checksum
**
**  Input:  Command Packet
**  Output: Char Checksum Value
*****************************************************************
unsigned char checksum_Compute(unsigned char command[]);

/*****************************************************************
**  checksum_Test
**
**  Compute Packet Checksum And Test Against Included Checksum
**
**  Input:  Command Packet
**  Output: 0 If Checksums Are Equal, Else 1
*****************************************************************
int checksum_Test(unsigned char command[]);

/*****************************************************************
** AccelStepper_Write
**
**  Parse command packet and write speed, direction, and number of steps
**
**  Input:  Command Packet
**  Output: None
*****************************************************************
void AccelStepper_Write(unsigned char command[]);
/*****************************************************************
**  SampleContinuosly
**
**  Returns several analog input points at once.
```

```
**
**  Input:  void
**  Output: void
*************************************************************
*************
void sampleContinously(void);

/*************************************************************
*************
**  finiteAcquisition
**
**  Returns the number of samples specified at the rate
    specified.
**
**  Input:  pin to sampe on, speed to sample at, number
    of samples
**  Output: void
*************************************************************
*************
void finiteAcquisition(int analogPin,float
acquisitionSpeed,int numberO
/*************************************************************
*************
**  lcd_print
**
**  Prints Data to the LCD With The Given Base
**
**  Input:  Command Packet
**  Output: None
*************************************************************
*************
void lcd_print(unsigned char command[]);
```