LVIFA_BASE

```
/*****************************************
*****************************************
 **
 **  LVFA_Firmware - Provides Basic Arduino
Sketch For Interfacing With LabVIEW.
 **
 **  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 Content
 **  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"

/*****************************************
*****************************************
 **  setup()
 **
 **  Initialize the Arduino and setup serial
communication.
 **
 **  Input:  None
 **  Output: None

*****************************************
*****************************************/
void setup()
{
  // Initialize Serial Port With The Default Baud Rate
  syncLV();

  // Place your custom setup code here

}

/*****************************************
*****************************************
 **  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 LabVIEW)


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

}
```

AFmotor.cpp

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


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

#include "AFMotor.h"

static uint8_t latch_state;

#if (MICROSTEPS == 8)
uint8_t microstepcurve[] = {0, 50, 98, 142, 180,
212, 236, 250, 255};
#elif (MICROSTEPS == 16)
uint8_t microstepcurve[] = {0, 25, 50, 74, 98, 120,
141, 162, 180, 197, 212, 225, 236, 244, 250, 253,
255};
#endif

AFMotorController::AFMotorController(void) {
}

void AFMotorController::enable(void) {
  // setup the latch
  /*
  LATCH_DDR |= _BV(LATCH);
  ENABLE_DDR |= _BV(ENABLE);
  CLK_DDR |= _BV(CLK);
  SER_DDR |= _BV(SER);
  */
  pinMode(MOTORLATCH, OUTPUT);
  pinMode(MOTORENABLE, OUTPUT);
  pinMode(MOTORDATA, OUTPUT);
  pinMode(MOTORCLK, OUTPUT);

  latch_state = 0;

  latch_tx();  // "reset"

  //ENABLE_PORT &= ~_BV(ENABLE); // enable
the chip outputs!
  digitalWrite(MOTORENABLE, LOW);
}


void AFMotorController::latch_tx(void) {
  uint8_t i;

  //LATCH_PORT &= ~_BV(LATCH);
  digitalWrite(MOTORLATCH, LOW);
  //SER_PORT &= ~_BV(SER);
  digitalWrite(MOTORDATA, LOW);

  for (i=0; i<8; i++) {
    //CLK_PORT &= ~_BV(CLK);
    digitalWrite(MOTORCLK, LOW);

    if (latch_state & _BV(7-i)) {
      //SER_PORT |= _BV(SER);
      digitalWrite(MOTORDATA, HIGH);
    } else {
      //SER_PORT &= ~_BV(SER);
      digitalWrite(MOTORDATA, LOW);
    }
    //CLK_PORT |= _BV(CLK);
    digitalWrite(MOTORCLK, HIGH);
  }
  //LATCH_PORT |= _BV(LATCH);
  digitalWrite(MOTORLATCH, HIGH);
}

static AFMotorController MC;


/******************************************
*
            MOTORS
******************************************
/
inline void initPWM1(uint8_t freq) {
#if defined(__AVR_ATmega8__) || \
    defined(__AVR_ATmega48__) || \
    defined(__AVR_ATmega88__) || \
    defined(__AVR_ATmega168__) || \
    defined(__AVR_ATmega328P__)
    // use PWM from timer2A on PB3 (Arduino pin
#11)
    TCCR2A |= _BV(COM2A1) | _BV(WGM20) |
_BV(WGM21); // fast PWM, turn on oc2a
    TCCR2B = freq & 0x7;
    OCR2A = 0;
#elif defined(__AVR_ATmega1280__) ||
defined(__AVR_ATmega2560__)
    // on arduino mega, pin 11 is now PB5 (OC1A)
    TCCR1A |= _BV(COM1A1) | _BV(WGM10); //
fast PWM, turn on oc1a
    TCCR1B = (freq & 0x7) | _BV(WGM12);
    OCR1A = 0;
#else
   #error "This chip is not supported!"
#endif
    pinMode(11, OUTPUT);
}

inline void setPWM1(uint8_t s) {
#if defined(__AVR_ATmega8__) || \
    defined(__AVR_ATmega48__) || \
    defined(__AVR_ATmega88__) || \
    defined(__AVR_ATmega168__) || \
    defined(__AVR_ATmega328P__)
    // use PWM from timer2A on PB3 (Arduino pin
#11)
```

```
    OCR2A = s;
#elif defined(__AVR_ATmega1280__) ||
defined(__AVR_ATmega2560__)
    // on arduino mega, pin 11 is now PB5 (OC1A)
    OCR1A = s;
#else
   #error "This chip is not supported!"
#endif
}

inline void initPWM2(uint8_t freq) {
#if defined(__AVR_ATmega8__) || \
    defined(__AVR_ATmega48__) || \
    defined(__AVR_ATmega88__) || \
    defined(__AVR_ATmega168__) || \
    defined(__AVR_ATmega328P__)
    // use PWM from timer2B (pin 3)
    TCCR2A |= _BV(COM2B1) | _BV(WGM20) |
_BV(WGM21); // fast PWM, turn on oc2b
    TCCR2B = freq & 0x7;
    OCR2B = 0;
#elif defined(__AVR_ATmega1280__) ||
defined(__AVR_ATmega2560__)
    // on arduino mega, pin 3 is now PE5 (OC3C)
    TCCR3A |= _BV(COM1C1) | _BV(WGM10); //
fast PWM, turn on oc3c
    TCCR3B = (freq & 0x7) | _BV(WGM12);
    OCR3C = 0;
#else
   #error "This chip is not supported!"
#endif

    pinMode(3, OUTPUT);
}

inline void setPWM2(uint8_t s) {
#if defined(__AVR_ATmega8__) || \
    defined(__AVR_ATmega48__) || \
    defined(__AVR_ATmega88__) || \
    defined(__AVR_ATmega168__) || \
    defined(__AVR_ATmega328P__)
    // use PWM from timer2A on PB3 (Arduino pin
#11)
    OCR2B = s;
#elif defined(__AVR_ATmega1280__) ||
defined(__AVR_ATmega2560__)
    // on arduino mega, pin 11 is now PB5 (OC1A)
    OCR3C = s;
#else
   #error "This chip is not supported!"
#endif
}

inline void initPWM3(uint8_t freq) {
#if defined(__AVR_ATmega8__) || \
    defined(__AVR_ATmega48__) || \
    defined(__AVR_ATmega88__) || \
    defined(__AVR_ATmega168__) || \
    defined(__AVR_ATmega328P__)
    // use PWM from timer0A / PD6 (pin 6)
    TCCR0A |= _BV(COM0A1) | _BV(WGM00) |
_BV(WGM01); // fast PWM, turn on OC0A
    //TCCR0B = freq & 0x7;
    OCR0A = 0;
#elif defined(__AVR_ATmega1280__) ||
defined(__AVR_ATmega2560__)
    // on arduino mega, pin 6 is now PH3 (OC4A)
    TCCR4A |= _BV(COM1A1) | _BV(WGM10); //
fast PWM, turn on oc4a
    TCCR4B = (freq & 0x7) | _BV(WGM12);
    //TCCR4B = 1 | _BV(WGM12);
    OCR4A = 0;
#else
   #error "This chip is not supported!"
#endif
    pinMode(6, OUTPUT);
}

inline void setPWM3(uint8_t s) {
#if defined(__AVR_ATmega8__) || \
    defined(__AVR_ATmega48__) || \
    defined(__AVR_ATmega88__) || \
    defined(__AVR_ATmega168__) || \
    defined(__AVR_ATmega328P__)
    // use PWM from timer0A on PB3 (Arduino pin
#6)
    OCR0A = s;
#elif defined(__AVR_ATmega1280__) ||
defined(__AVR_ATmega2560__)
    // on arduino mega, pin 6 is now PH3 (OC4A)
    OCR4A = s;
#else
   #error "This chip is not supported!"
#endif
}



inline void initPWM4(uint8_t freq) {
#if defined(__AVR_ATmega8__) || \
    defined(__AVR_ATmega48__) || \
    defined(__AVR_ATmega88__) || \
    defined(__AVR_ATmega168__) || \
    defined(__AVR_ATmega328P__)
    // use PWM from timer0B / PD5 (pin 5)
    TCCR0A |= _BV(COM0B1) | _BV(WGM00) |
_BV(WGM01); // fast PWM, turn on oc0a
    //TCCR0B = freq & 0x7;
    OCR0B = 0;
#elif defined(__AVR_ATmega1280__) ||
defined(__AVR_ATmega2560__)
    // on arduino mega, pin 5 is now PE3 (OC3A)
    TCCR3A |= _BV(COM1A1) | _BV(WGM10); //
fast PWM, turn on oc3a
    TCCR3B = (freq & 0x7) | _BV(WGM12);
    //TCCR4B = 1 | _BV(WGM12);
    OCR3A = 0;
#else
   #error "This chip is not supported!"
#endif
```

```
    pinMode(5, OUTPUT);
}

inline void setPWM4(uint8_t s) {
#if defined(__AVR_ATmega8__) || \
    defined(__AVR_ATmega48__) || \
    defined(__AVR_ATmega88__) || \
    defined(__AVR_ATmega168__) || \
    defined(__AVR_ATmega328P__)
    // use PWM from timer0A on PB3 (Arduino pin
#6)
    OCR0B = s;
#elif defined(__AVR_ATmega1280__) ||
defined(__AVR_ATmega2560__)
    // on arduino mega, pin 6 is now PH3 (OC4A)
    OCR3A = s;
#else
   #error "This chip is not supported!"
#endif
}

AF_DCMotor::AF_DCMotor(uint8_t num, uint8_t
freq) {
  motornum = num;
  pwmfreq = freq;

  MC.enable();

  switch (num) {
  case 1:
    latch_state &= ~_BV(MOTOR1_A) &
~_BV(MOTOR1_B); // set both motor pins to 0
    MC.latch_tx();
    initPWM1(freq);
    break;
  case 2:
    latch_state &= ~_BV(MOTOR2_A) &
~_BV(MOTOR2_B); // set both motor pins to 0
    MC.latch_tx();
    initPWM2(freq);
    break;
  case 3:
    latch_state &= ~_BV(MOTOR3_A) &
~_BV(MOTOR3_B); // set both motor pins to 0
    MC.latch_tx();
    initPWM3(freq);
    break;
  case 4:
    latch_state &= ~_BV(MOTOR4_A) &
~_BV(MOTOR4_B); // set both motor pins to 0
    MC.latch_tx();
    initPWM4(freq);
    break;
  }
}

void AF_DCMotor::run(uint8_t cmd) {
  uint8_t a, b;
  switch (motornum) {
  case 1:
    a = MOTOR1_A; b = MOTOR1_B; break;
  case 2:
    a = MOTOR2_A; b = MOTOR2_B; break;
  case 3:
    a = MOTOR3_A; b = MOTOR3_B; break;
  case 4:
    a = MOTOR4_A; b = MOTOR4_B; break;
  default:
    return;
  }

  switch (cmd) {
  case FORWARD:
    latch_state |= _BV(a);
    latch_state &= ~_BV(b);
    MC.latch_tx();
    break;
  case BACKWARD:
    latch_state &= ~_BV(a);
    latch_state |= _BV(b);
    MC.latch_tx();
    break;
  case RELEASE:
    latch_state &= ~_BV(a);
    latch_state &= ~_BV(b);
    MC.latch_tx();
    break;
  }
}

void AF_DCMotor::setSpeed(uint8_t speed) {
  switch (motornum) {
  case 1:
    setPWM1(speed); break;
  case 2:
    setPWM2(speed); break;
  case 3:
    setPWM3(speed); break;
  case 4:
    setPWM4(speed); break;
  }
}

/*****************************************
*
            STEPPERS
*****************************************
/

AF_Stepper::AF_Stepper(uint16_t steps, uint8_t
num) {
  MC.enable();

  revsteps = steps;
  steppernum = num;
  currentstep = 0;

  if (steppernum == 1) {
    latch_state &= ~_BV(MOTOR1_A) &
~_BV(MOTOR1_B) &
```

```
    ~_BV(MOTOR2_A) & ~_BV(MOTOR2_B); //
all motor pins to 0
   MC.latch_tx();

   // enable both H bridges
   pinMode(11, OUTPUT);
   pinMode(3, OUTPUT);
   digitalWrite(11, HIGH);
   digitalWrite(3, HIGH);

   // use PWM for microstepping support
   initPWM1(MOTOR12_64KHZ);
   initPWM2(MOTOR12_64KHZ);
   setPWM1(255);
   setPWM2(255);

  } else if (steppernum == 2) {
   latch_state &= ~_BV(MOTOR3_A) &
~_BV(MOTOR3_B) &
    ~_BV(MOTOR4_A) & ~_BV(MOTOR4_B); //
all motor pins to 0
   MC.latch_tx();

   // enable both H bridges
   pinMode(5, OUTPUT);
   pinMode(6, OUTPUT);
   digitalWrite(5, HIGH);
   digitalWrite(6, HIGH);

   // use PWM for microstepping support
   // use PWM for microstepping support
   initPWM3(1);
   initPWM4(1);
   setPWM3(255);
   setPWM4(255);
  }
}

void AF_Stepper::setSpeed(uint16_t rpm) {
  usperstep = 60000000 / ((uint32_t)revsteps *
(uint32_t)rpm);
  steppingcounter = 0;
}

void AF_Stepper::release(void) {
  if (steppernum == 1) {
   latch_state &= ~_BV(MOTOR1_A) &
~_BV(MOTOR1_B) &
    ~_BV(MOTOR2_A) & ~_BV(MOTOR2_B); //
all motor pins to 0
   MC.latch_tx();
  } else if (steppernum == 2) {
   latch_state &= ~_BV(MOTOR3_A) &
~_BV(MOTOR3_B) &
    ~_BV(MOTOR4_A) & ~_BV(MOTOR4_B); //
all motor pins to 0
   MC.latch_tx();
  }
}

void AF_Stepper::step(uint16_t steps, uint8_t dir,
uint8_t style) {
  uint32_t uspers = usperstep;
  uint8_t ret = 0;

  if (style == INTERLEAVE) {
   uspers /= 2;
  }
 else if (style == MICROSTEP) {
   uspers /= MICROSTEPS;
   steps *= MICROSTEPS;
#ifdef MOTORDEBUG
   Serial.print("steps = "); Serial.println(steps,
DEC);
#endif
  }

  while (steps--) {
   ret = onestep(dir, style);
   delay(uspers/1000); // in ms
   steppingcounter += (uspers % 1000);
   if (steppingcounter >= 1000) {
    delay(1);
    steppingcounter -= 1000;
   }
  }
  if (style == MICROSTEP) {
   while ((ret != 0) && (ret != MICROSTEPS)) {
    ret = onestep(dir, style);
    delay(uspers/1000); // in ms
    steppingcounter += (uspers % 1000);
    if (steppingcounter >= 1000) {
        delay(1);
        steppingcounter -= 1000;
    }
   }
  }
}

uint8_t AF_Stepper::onestep(uint8_t dir, uint8_t
style) {
  uint8_t a, b, c, d;
  uint8_t ocrb, ocra;

  ocra = ocrb = 255;

  if (steppernum == 1) {
   a = _BV(MOTOR1_A);
   b = _BV(MOTOR2_A);
   c = _BV(MOTOR1_B);
   d = _BV(MOTOR2_B);
  } else if (steppernum == 2) {
   a = _BV(MOTOR3_A);
   b = _BV(MOTOR4_A);
   c = _BV(MOTOR3_B);
   d = _BV(MOTOR4_B);
  } else {
   return 0;
  }
```

```
  // next determine what sort of stepping procedure
we're up to
  if (style == SINGLE) {
    if ((currentstep/(MICROSTEPS/2)) % 2) { //
we're at an odd step, weird
      if (dir == FORWARD) {
           currentstep += MICROSTEPS/2;
      }
      else {
           currentstep -= MICROSTEPS/2;
      }
    } else {          // go to the next even step
      if (dir == FORWARD) {
           currentstep += MICROSTEPS;
      }
      else {
           currentstep -= MICROSTEPS;
      }
    }
  } else if (style == DOUBLE) {
    if (! (currentstep/(MICROSTEPS/2) % 2)) { //
we're at an even step, weird
      if (dir == FORWARD) {
           currentstep += MICROSTEPS/2;
      } else {
           currentstep -= MICROSTEPS/2;
      }
    } else {          // go to the next odd step
      if (dir == FORWARD) {
           currentstep += MICROSTEPS;
      } else {
           currentstep -= MICROSTEPS;
      }
    }
  } else if (style == INTERLEAVE) {
    if (dir == FORWARD) {
       currentstep += MICROSTEPS/2;
    } else {
       currentstep -= MICROSTEPS/2;
    }
  }

  if (style == MICROSTEP) {
    if (dir == FORWARD) {
      currentstep++;
    } else {
      // BACKWARDS
      currentstep--;
    }

    currentstep += MICROSTEPS*4;
    currentstep %= MICROSTEPS*4;

    ocra = ocrb = 0;
    if ( (currentstep >= 0) && (currentstep <
MICROSTEPS)) {
      ocra = microstepcurve[MICROSTEPS -
currentstep];
      ocrb = microstepcurve[currentstep];
    } else if  ( (currentstep >= MICROSTEPS) &&
(currentstep < MICROSTEPS*2)) {
      ocra = microstepcurve[currentstep -
MICROSTEPS];
      ocrb = microstepcurve[MICROSTEPS*2 -
currentstep];
    } else if  ( (currentstep >= MICROSTEPS*2) &&
(currentstep < MICROSTEPS*3)) {
      ocra = microstepcurve[MICROSTEPS*3 -
currentstep];
      ocrb = microstepcurve[currentstep -
MICROSTEPS*2];
    } else if  ( (currentstep >= MICROSTEPS*3) &&
(currentstep < MICROSTEPS*4)) {
      ocra = microstepcurve[currentstep -
MICROSTEPS*3];
      ocrb = microstepcurve[MICROSTEPS*4 -
currentstep];
    }
  }

  currentstep += MICROSTEPS*4;
  currentstep %= MICROSTEPS*4;

#ifdef MOTORDEBUG
  Serial.print("current step: ");
Serial.println(currentstep, DEC);
  Serial.print(" pwmA = "); Serial.print(ocra, DEC);
  Serial.print(" pwmB = "); Serial.println(ocrb,
DEC);
#endif

  if (steppernum == 1) {
    setPWM1(ocra);
    setPWM2(ocrb);
  } else if (steppernum == 2) {
    setPWM3(ocra);
    setPWM4(ocrb);
  }


  // release all
  latch_state &= ~a & ~b & ~c & ~d; // all motor
pins to 0

  //Serial.println(step, DEC);
  if (style == MICROSTEP) {
    if ((currentstep >= 0) && (currentstep <
MICROSTEPS))
      latch_state |= a | b;
    if ((currentstep >= MICROSTEPS) &&
(currentstep < MICROSTEPS*2))
      latch_state |= b | c;
    if ((currentstep >= MICROSTEPS*2) &&
(currentstep < MICROSTEPS*3))
      latch_state |= c | d;
    if ((currentstep >= MICROSTEPS*3) &&
(currentstep < MICROSTEPS*4))
      latch_state |= d | a;
  } else {
```

```
  switch (currentstep/(MICROSTEPS/2)) {
  case 0:
    latch_state |= a; // energize coil 1 only
    break;
  case 1:
    latch_state |= a | b; // energize coil 1+2
    break;
  case 2:
    latch_state |= b; // energize coil 2 only
    break;
  case 3:
    latch_state |= b | c; // energize coil 2+3
    break;
  case 4:
    latch_state |= c; // energize coil 3 only
    break;
  case 5:
    latch_state |= c | d; // energize coil 3+4
    break;
  case 6:
    latch_state |= d; // energize coil 4 only
    break;
  case 7:
    latch_state |= d | a; // energize coil 1+4
    break;
   }
 }


 MC.latch_tx();
 return currentstep;
}
```

AFmotor.h

```
// 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);
    friend class 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

```
// 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 current time and next step time wrap
//        || ((time < _lastRunTime) && (time >
(0xFFFFFFFF-(_lastStepTime+_stepInterval)))) )  //
check if only current time has wrapped

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

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


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

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

    {
        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 works with + and - numbers)

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

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

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

long AccelStepper::currentPosition()
{
    return _currentPos;
}

// Useful during initialisations or after initial
positioning
void AccelStepper::setCurrentPosition(long
position)
{
    _targetPos = _currentPos = position;
    computeNewSpeed(); // Expect speed of 0
}

void AccelStepper::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 distance
    // 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 extra calculations below
        {
            // 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 extra calculations below
        {
            // 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 to the target
position
// You must call this at least once per step,
preferably in your main loop
// 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, uint8_t pin3, uint8_t pin4)
{
    _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_t step)
{
    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_t step)
{
    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_t step)
{
    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_t step)
{
    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);
    }
}

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

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

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

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

Accelstepper.h

```
// AccelStepper.h
//
/// \mainpage AccelStepper library for Arduino
///
/// This is the Arduino AccelStepper library.
/// It provides an object-oriented interface for 2 or 4 pin stepper motors.
///
/// The standard Arduino IDE includes the Stepper library
/// (http://arduino.cc/en/Reference/Stepper) for stepper motors. It is
/// perfectly adequate for simple, single motor applications.
///
/// AccelStepper significantly improves on the standard Arduino Stepper library in several ways:
/// \li Supports acceleration and deceleration
/// \li Supports multiple simultaneous steppers, with independent concurrent stepping on each stepper
/// \li API functions never delay() or block
/// \li Supports 2 and 4 wire steppers, plus 4 wire half steppers.
/// \li Supports alternate stepping functions to enable support of AFMotor (https://github.com/adafruit/Adafruit-Motor-Shield-library)
/// \li Supports stepper drivers such as the Sparkfun EasyDriver (based on 3967 driver chip)
/// \li Very slow speeds are supported
/// \li Extensive API
/// \li Subclass support
///
/// The latest version of this documentation can be downloaded from
/// http://www.open.com.au/mikem/arduino/AccelStepper
///
/// Example Arduino programs are included to show the main modes of use.
///
/// The version of the package that this documentation refers to can be downloaded
/// from http://www.open.com.au/mikem/arduino/AccelStepper/AccelStepper-1.9.zip
/// You can find the latest version at http://www.open.com.au/mikem/arduino/AccelStepper
///
/// Tested on Arduino Diecimila and Mega with arduino-0018 & arduino-0021
/// on OpenSuSE 11.1 and avr-libc-1.6.1-1.15,
/// cross-avr-binutils-2.19-9.1, cross-avr-gcc-4.1.3_20080612-26.5.
///
/// \par Installation
/// Install in the usual way: unzip the distribution zip file to the libraries
/// sub-folder of your sketchbook.
///
/// This software is Copyright (C) 2010 Mike McCauley. Use is subject to license
/// conditions. The main licensing options available are GPL V2 or Commercial:
///
/// \par Open Source Licensing GPL V2
/// This is the appropriate option if you want to share the source code of your
/// application with everyone you distribute it to, and you also want to give them
/// the right to share who uses it. If you wish to use this software under Open
/// Source Licensing, you must contribute all your source code to the open source
/// community in accordance with the GPL Version 2 when your application is
/// distributed. See http://www.gnu.org/copyleft/gpl.html
///
/// \par Commercial Licensing
/// This is the appropriate option if you are creating proprietary applications
/// and you are not prepared to distribute and share the source code of your
/// application. Contact info@open.com.au for details.
///
/// \par Revision History
/// \version 1.0 Initial release
///
/// \version 1.1 Added speed() function to get the current speed.
/// \version 1.2 Added runSpeedToPosition() submitted by Gunnar Arndt.
/// \version 1.3 Added support for stepper drivers (ie with Step and Direction inputs) with _pins == 1
/// \version 1.4 Added functional contructor to support AFMotor, contributed by Limor, with example sketches.
/// \version 1.5 Improvements contributed by Peter Mousley: Use of microsecond steps and other speed improvements
///              to increase max stepping speed to about 4kHz. New option for user to set the min allowed pulse width.
///              Added checks for already running at max speed and skip further calcs if so.
/// \version 1.6 Fixed a problem with wrapping of microsecond stepping that could cause stepping to hang.
///              Reported by Sandy Noble.
///              Removed redundant _lastRunTime member.
/// \version 1.7 Fixed a bug where setCurrentPosition() did always work as expected. Reported by Peter Linhart.
```

```
///          Reported by Sandy Noble.
///          Removed redundant _lastRunTime
member.
/// \version 1.8 Added support for 4 pin half-
steppers, requested by Harvey Moon
/// \version 1.9 setCurrentPosition() now also sets
motor speed to 0.
///
///
/// \author  Mike McCauley (mikem@open.com.au)
// Copyright (C) 2009 Mike McCauley
// $Id: AccelStepper.h,v 1.5 2011/03/21 00:42:15
mikem Exp mikem $

#ifndef AccelStepper_h
#define AccelStepper_h

#include <stdlib.h>
 #if defined(ARDUINO) && ARDUINO >= 100
  #include <pins_arduino.h>
  #else
  #include <wiring.h>
  #endif



// These defs cause trouble on some versions of
Arduino
#undef round

//////////////////////////////////////////////////////////////////////
/// \class AccelStepper AccelStepper.h
<AccelStepper.h>
/// \brief Support for stepper motors with
acceleration etc.
///
/// This defines a single 2 or 4 pin stepper motor, or
stepper moter with fdriver chip, with optional
/// acceleration, deceleration, absolute positioning
commands etc. Multiple
/// simultaneous steppers are supported, all moving
/// at different speeds and accelerations.
///
/// \par Operation
/// This module operates by computing a step time in
microseconds. The step
/// time is recomputed after each step and after speed
and acceleration
/// parameters are changed by the caller. The time of
each step is recorded in
/// microseconds. The run() function steps the motor
if a new step is due.
/// The run() function must be called frequently until
the motor is in the
/// desired position, after which time run() will do
nothing.
///
/// \par Positioning
/// Positions are specified by a signed long integer.
At
/// construction time, the current position of the
motor is consider to be 0. Positive
/// positions are clockwise from the initial position;
negative positions are
/// anticlockwise. The curent position can be altered
for instance after
/// initialization positioning.
///
/// \par Caveats
/// This is an open loop controller: If the motor stalls
or is oversped,
/// AccelStepper will not have a correct
/// idea of where the motor really is (since there is no
feedback of the motor's
/// real position. We only know where we _think_ it
is, relative to the
/// initial starting point).
///
/// The fastest motor speed that can be reliably
supported is 4000 steps per
/// second (4 kHz) at a clock frequency of 16 MHz.
However, any speed less than that
/// down to very slow speeds (much less than one
per second) are also supported,
/// provided the run() function is called frequently
enough to step the motor
/// whenever required for the speed set.
class AccelStepper
{
public:
    /// Constructor. You can have multiple
simultaneous steppers, all moving
    /// at different speeds and accelerations, provided
you call their run()
    /// functions at frequent enough intervals. Current
Position is set to 0, target
    /// position is set to 0. MaxSpeed and
Acceleration default to 1.0.
    /// The motor pins will be initialised to OUTPUT
mode during the
    /// constructor by a call to enableOutputs().
    /// \param[in] pins Number of pins to interface to.
1, 2 or 4 are
    /// supported. 1 means a stepper driver (with Step
and Direction pins)
    /// 2 means a 2 wire stepper. 4 means a 4 wire
stepper. 8 means a 4 wire half stepper
    /// Defaults to 4 pins.
    /// \param[in] pin1 Arduino digital pin number for
motor pin 1. Defaults
    /// to pin 2. For a driver (pins==1), this is the Step
input to the driver. Low to high transition means to
step)
    /// \param[in] pin2 Arduino digital pin number for
motor pin 2. Defaults
    /// to pin 3. For a driver (pins==1), this is the
Direction input the driver. High means forward.
    /// \param[in] pin3 Arduino digital pin number for
motor pin 3. Defaults
    /// to pin 4.
```

```
    /// \param[in] pin4 Arduino digital pin number for motor pin 4. Defaults
    /// to pin 5.
    AccelStepper(uint8_t pins = 4, uint8_t pin1 = 2, uint8_t pin2 = 3, uint8_t pin3 = 4, uint8_t pin4 = 5);

    /// Alternate Constructor which will call your own functions for forward and backward steps.
    /// You can have multiple simultaneous steppers, all moving
    /// at different speeds and accelerations, provided you call their run()
    /// functions at frequent enough intervals. Current Position is set to 0, target
    /// position is set to 0. MaxSpeed and Acceleration default to 1.0.
    /// Any motor initialization should happen before hand, no pins are used or initialized.
    /// \param[in] forward void-returning procedure that will make a forward step
    /// \param[in] backward void-returning procedure that will make a backward step
    AccelStepper(void (*forward)(), void (*backward)());

    /// Set the target position. The run() function will try to move the motor
    /// from the current position to the target position set by the most
    /// recent call to this function.
    /// \param[in] absolute The desired absolute position. Negative is
    /// anticlockwise from the 0 position.
    void    moveTo(long absolute);

    /// Set the target position relative to the current position
    /// \param[in] relative The desired position relative to the current position. Negative is
    /// anticlockwise from the current position.
    void    move(long relative);

    /// Poll the motor and step it if a step is due, implementing
    /// accelerations and decelerations to achive the ratget position. You must call this as
    /// fequently as possible, but at least once per minimum step interval,
    /// preferably in your main loop.
    /// \return true if the motor is at the target position.
    boolean run();

    /// Poll the motor and step it if a step is due, implmenting a constant
    /// speed as set by the most recent call to setSpeed().
    /// \return true if the motor was stepped.
    boolean runSpeed();

    /// Sets the maximum permitted speed. the run() function will accelerate
    /// up to the speed set by this function.
    /// \param[in] speed The desired maximum speed in steps per second. Must
    /// be > 0. Speeds of more than 1000 steps per second are unreliable.
    void    setMaxSpeed(float speed);

    /// Sets the acceleration and deceleration parameter.
    /// \param[in] acceleration The desired acceleration in steps per second
    /// per second. Must be > 0.
    void    setAcceleration(float acceleration);

    /// Sets the desired constant speed for use with runSpeed().
    /// \param[in] speed The desired constant speed in steps per
    /// second. Positive is clockwise. Speeds of more than 1000 steps per
    /// second are unreliable. Very slow speeds may be set (eg 0.00027777 for
    /// once per hour, approximately. Speed accuracy depends on the Arduino
    /// crystal. Jitter depends on how frequently you call the runSpeed() function.
    void    setSpeed(float speed);

    /// The most recently set speed
    /// \return the most recent speed in steps per second
    float   speed();

    /// The distance from the current position to the target position.
    /// \return the distance from the current position to the target position
    /// in steps. Positive is clockwise from the current position.
    long    distanceToGo();

    /// The most recently set target position.
    /// \return the target position
    /// in steps. Positive is clockwise from the 0 position.
    long    targetPosition();


    /// The currently motor position.
    /// \return the current motor position
    /// in steps. Positive is clockwise from the 0 position.
    long    currentPosition();

    /// Resets the current position of the motor, so that wherever the motor
    /// happens to be right now is considered to be the new 0 position. Useful
```

```
    /// for setting a zero position on a stepper after an initial hardware
    /// positioning move.
    /// Has the side effect of setting the current motor speed to 0.
    /// \param[in] position The position in steps of wherever the motor
    /// happens to be right now.
    void    setCurrentPosition(long position);

    /// Moves the motor to the target position and blocks until it is at
    /// position. Dont use this in event loops, since it blocks.
    void    runToPosition();

    /// Runs at the currently selected speed until the target position is reached
    /// Does not implement accelerations.
    boolean runSpeedToPosition();

    /// Moves the motor to the new target position and blocks until it is at
    /// position. Dont use this in event loops, since it blocks.
    /// \param[in] position The new target position.
    void    runToNewPosition(long position);

    /// Disable motor pin outputs by setting them all LOW
    /// Depending on the design of your electronics this may turn off
    /// the power to the motor coils, saving power.
    /// This is useful to support Arduino low power modes: disable the outputs
    /// during sleep and then reenable with enableOutputs() before stepping
    /// again.
    void    disableOutputs();

    /// Enable motor pin outputs by setting the motor pins to OUTPUT
    /// mode. Called automatically by the constructor.
    void    enableOutputs();

    /// Sets the minimum pulse width allowed by the stepper driver.
    /// \param[in] minWidth The minimum pulse width in microseconds.
    void    setMinPulseWidth(unsigned int minWidth);

protected:

    /// Forces the library to compute a new instantaneous speed and set that as
    /// the current speed. Calls
    /// desiredSpeed(), which can be overridden by subclasses. It is called by
    /// the library:
    /// \li  after each step
    /// \li  after change to maxSpeed through setMaxSpeed()
    /// \li  after change to acceleration through setAcceleration()
    /// \li  after change to target position (relative or absolute) through
    /// move() or moveTo()
    void          computeNewSpeed();

    /// Called to execute a step. Only called when a new step is
    /// required. Subclasses may override to implement new stepping
    /// interfaces. The default calls step1(), step2(), step4() or step8() depending on the
    /// number of pins defined for the stepper.
    /// \param[in] step The current step phase number (0 to 7)
    virtual void   step(uint8_t step);

    /// Called to execute a step using stepper functions (pins = 0) Only called when a new step is
    /// required. Calls _forward() or _backward() to perform the step
    virtual void   step0(void);

    /// Called to execute a step on a stepper drover (ie where pins == 1). Only called when a new step is
    /// required. Subclasses may override to implement new stepping
    /// interfaces. The default sets or clears the outputs of Step pin1 to step,
    /// and sets the output of _pin2 to the desired direction. The Step pin (_pin1) is pulsed for 1 microsecond
    /// which is the minimum STEP pulse width for the 3967 driver.
    /// \param[in] step The current step phase number (0 to 7)
    virtual void   step1(uint8_t step);

    /// Called to execute a step on a 2 pin motor. Only called when a new step is
    /// required. Subclasses may override to implement new stepping
    /// interfaces. The default sets or clears the outputs of pin1 and pin2
    /// \param[in] step The current step phase number (0 to 7)
    virtual void   step2(uint8_t step);

    /// Called to execute a step on a 4 pin motor. Only called when a new step is
    /// required. Subclasses may override to implement new stepping
    /// interfaces. The default sets or clears the outputs of pin1, pin2,
    /// pin3, pin4.
```

```
    /// \param[in] step The current step phase number
(0 to 7)
    virtual void    step4(uint8_t step);

    /// Called to execute a step on a 4 pin half-steper
motor. Only called when a new step is
    /// required. Subclasses may override to
implement new stepping
    /// interfaces. The default sets or clears the outputs
of pin1, pin2,
    /// pin3, pin4.
    /// \param[in] step The current step phase number
(0 to 7)
    virtual void    step8(uint8_t step);

    /// Compute and return the desired speed. The
default algorithm uses
    /// maxSpeed, acceleration and the current speed
to set a new speed to
    /// move the motor from teh current position to the
target
    /// position. Subclasses may override this to
provide an alternate
    /// algorithm (but do not block). Called by
computeNewSpeed whenever a new speed neds to
be
    /// computed.
    virtual float  desiredSpeed();

private:
    /// Number of pins on the stepper motor. Permits
2 or 4. 2 pins is a
    /// bipolar, and 4 pins is a unipolar.
    uint8_t        _pins;          // 2 or 4

    /// Arduino pin number for the 2 or 4 pins
required to interface to the
    /// stepper motor.
    uint8_t        _pin1, _pin2, _pin3, _pin4;

    /// The current absolution position in steps.
    long           _currentPos;    // Steps

    /// The target position in steps. The AccelStepper
library will move the
    /// motor from teh _currentPos to the _targetPos,
taking into account the
    /// max speed, acceleration and deceleration
    long           _targetPos;     // Steps

    /// The current motos speed in steps per second
    /// Positive is clockwise
    float          _speed;         // Steps per second

    /// The maximum permitted speed in steps per
second. Must be > 0.
    float          _maxSpeed;

    /// The acceleration to use to accelerate or
decelerate the motor in steps
    /// per second per second. Must be > 0
    float          _acceleration;

    /// The current interval between steps in
microseconds
    unsigned long  _stepInterval;

    /// The last run time (when runSpeed() was last
called) in microseconds
//    unsigned long  _lastRunTime;

    /// The last step time in microseconds
    unsigned long  _lastStepTime;

    /// The minimum allowed pulse width in
microseconds
    unsigned int   _minPulseWidth;

    // The pointer to a forward-step procedure
    void (*_forward)();

    // The pointer to a backward-step procedure
    void (*_backward)();
};

#endif
```

LABVIEWInterface.h

```
/*****************************************
*****************************************
 **
 **
 **  LVFA_Firmware - Provides Functions For
Interfacing With The Arduino Uno
 **
 **  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 Content
 **  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 9600    // Defines
The Default Serial Baud Rate (This must match the
baud rate specifid in LabVIEW)
#else
#define DEFAULTBAUDRATE 115200
#endif
#define MODE_DEFAULT 0          // Defines
Arduino Modes (Currently Not Used)
#define COMMANDLENGTH 15        // Defines
The Number Of Bytes In A Single LabVIEW
Command (This must match the packet size specifid
in LabVIEW)
#define STEPPER_SUPPORT 1       // Defines
Whether The Stepper Library Is Included -
Comment This Line To Exclude Stepper Support


// Declare Variables
unsigned char
currentCommand[COMMANDLENGTH];    // The
Current Command For The Arduino To Process
//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 (Unimplemented)
**
**  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 outputs.
**
**  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 are stored in
sevenSegmentPins array.
**
**  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_Configure
**
**  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 data byte.
*****************************************
*****************************************/
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 to travel
**
**  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 numberOfSamples );
/*****************************************
*****************************************
** lcd_print
**
** Prints Data to the LCD With The Given Base
**
** Input: Command Packet
** Output: None
*****************************************
*****************************************/
void lcd_print(unsigned char command[]);
```

LABVIEWInterface

```
/*****************************************
*****************************************
 **
 ** LVIFA_Firmware - Provides Functions For
Interfacing With The Arduino Uno
 **
 ** 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 Content
 ** Written By Sam Kristoff And Available At
www.ni.com/arduino.
 **

*****************************************
*****************************************/

#include <Wire.h>
#include <SPI.h>
#include <LiquidCrystal.h>


/*****************************************
*****************************************
 ** Optionally Include And Configure Stepper
Support

*****************************************
*****************************************/
#ifdef STEPPER_SUPPORT

  // Stepper Modifications
  #include "AFMotor.h"
  #include "AccelStepper.h"

  // Adafruit shield
  AF_Stepper motor1(200, 1);
  AF_Stepper motor2(200, 2);

  // you can change these to DOUBLE or
INTERLEAVE or MICROSTEP
  // wrappers for the first motor
  void forwardstep1() {
    motor1.onestep(FORWARD, SINGLE);
  }
  void backwardstep1() {
    motor1.onestep(BACKWARD, SINGLE);
  }
  // wrappers for the second motor
  void forwardstep2() {
    motor2.onestep(FORWARD, SINGLE);
  }
  void backwardstep2() {
    motor2.onestep(BACKWARD, SINGLE);
```

```
  }

  AccelStepper steppers[8];  //Create array of 8
stepper objects

#endif

// Variables

unsigned int retVal;
int sevenSegmentPins[8];
int currentMode;
unsigned int freq;
unsigned long duration;
int i2cReadTimeouts = 0;
char spiBytesToSend = 0;
char spiBytesSent = 0;
char spiCSPin = 0;
char spiWordSize = 0;
Servo *servos;
byte customChar[8];
LiquidCrystal lcd(0,0,0,0,0,0,0);
// Sets the mode of the Arduino (Reserved For
Future Use)
void setMode(int mode)
{
  currentMode = mode;
}

// Checks for new commands from LabVIEW and
processes them if any exists.
int checkForCommand(void)
{
#ifdef STEPPER_SUPPORT
  // Call run function as fast as possible to keep
motors turning
  for (int i=0; i<8; i++){
    steppers[i].run();
  }
#endif

  int bufferBytes = Serial.available();

  if(bufferBytes >= COMMANDLENGTH)
  {
    // New Command Ready, Process It
    // Build Command From Serial Buffer
    for(int i=0; i<COMMANDLENGTH; i++)
    {
      currentCommand[i] = Serial.read();
    }
    processCommand(currentCommand);
    return 1;
  }
  else
  {
    return 0;
  }
}

// Processes a given command
void processCommand(unsigned char command[])
{
  // Determine Command
  if(command[0] == 0xFF &&
checksum_Test(command) == 0)
  {
    switch(command[1])
    {

/*****************************************
*************************************
    ** LIFA Maintenance Commands

*****************************************
*************************************/
    case 0x00:     // Sync Packet
      Serial.print("sync");
      Serial.flush();
      break;
    case 0x01:    // Flush Serial Buffer
      Serial.flush();
      break;


/*****************************************
*************************************
    ** Low Level - Digital I/O Commands

*****************************************
*************************************/
    case 0x02:    // Set Pin As Input Or Output
      pinMode(command[2], command[3]);
      Serial.write('0');
      break;
    case 0x03:    // Write Digital Pin
      digitalWrite(command[2], command[3]);
       Serial.write('0');
      break;
    case 0x04:    // Write Digital Port 0
      writeDigitalPort(command);
       Serial.write('0');
      break;
    case 0x05:    //Tone
      freq = ( (command[3]<<8) + command[4]);
      duration=(command[8]+  (command[7]<<8)+
(command[6]<<16)+(command[5]<<24));

      if(freq > 0)
      {
        tone(command[2], freq, duration);
      }
      else
      {
        noTone(command[2]);
      }
       Serial.write('0');
      break;
    case 0x06:    // Read Digital Pin
      retVal = digitalRead(command[2]);
```

```
    Serial.write(retVal);
    break;
  case 0x07:    // Digital Read Port
    retVal = 0x0000;
    for(int i=0; i <=13; i++)
    {
     if(digitalRead(i))
     {
       retVal += (1<<i);
     }
    }
    Serial.write( (retVal & 0xFF));
    Serial.write( (retVal >> 8));
    break;


/*****************************************
******************************************
   ** Low Level - Analog Commands

*****************************************
******************************************/
  case 0x08:    // Read Analog Pin
    retVal = analogRead(command[2]);
    Serial.write( (retVal >> 8));
    Serial.write( (retVal & 0xFF));
    break;
  case 0x09:    // Analog Read Port
    analogReadPort();
    break;


/*****************************************
******************************************
   ** Low Level - PWM Commands

*****************************************
******************************************/
  case 0x0A:    // PWM Write Pin
    analogWrite(command[2], command[3]);
     Serial.write('0');
    break;
  case 0x0B:    // PWM Write 3 Pins
    analogWrite(command[2], command[5]);
    analogWrite(command[3], command[6]);
    analogWrite(command[4], command[7]);
     Serial.write('0');
    break;


/*****************************************
******************************************
   ** Sensor Specific Commands

*****************************************
******************************************/
  case 0x0C:    // Configure Seven Segment Display
    sevenSegment_Config(command);
     Serial.write('0');
    break;
  case 0x0D:    // Write To Seven Segment Display
    sevenSegment_Write(command);
     Serial.write('0');
    break;


/*****************************************
******************************************
   **  I2C

*****************************************
******************************************/
  case 0x0E:    // Initialize I2C
    Wire.begin();
     Serial.write('0');
    break;
  case 0x0F:    // Send I2C Data
    Wire.beginTransmission(command[3]);
    for(int i=0; i<command[2]; i++)
    {
     #if defined(ARDUINO) && ARDUINO >=
100
       Wire.write(command[i+4]);
     #else
       Wire.send(command[i+4]);
     #endif

    }
    Wire.endTransmission();
     Serial.write('0');
    break;
  case 0x10:    // I2C Read
    i2cReadTimeouts = 0;
    Wire.requestFrom(command[3], command[2]);
    while(Wire.available() < command[2])
    {
     i2cReadTimeouts++;
     if(i2cReadTimeouts > 100)
     {
      return;
     }
     else
     {
      delay(1);
     }
    }

    for(int i=0; i<command[2]; i++)
    {
     #if defined(ARDUINO) && ARDUINO >=
100
       Serial.write(Wire.read());
     #else
       Serial.write(Wire.receive());
     #endif

    }
    break;
```

```
/*****************************************
*****************************************
   ** SPI

*****************************************
*****************************************/
   case 0x11:    // SPI Init
     SPI.begin();
      Serial.write('0');
     break;
   case 0x12:    // SPI Set Bit Order (MSB LSB)
     if(command[2] == 0)
     {
       SPI.setBitOrder(LSBFIRST);
     }
     else
     {
       SPI.setBitOrder(MSBFIRST);
     }
      Serial.write('0');
     break;
   case 0x13:    // SPI Set Clock Divider
     spi_setClockDivider(command[2]);
      Serial.write('0');
     break;
   case 0x14:    // SPI Set Data Mode
     switch(command[2])
     {
     case 0:
       SPI.setDataMode(SPI_MODE0);
       break;
     case 1:
       SPI.setDataMode(SPI_MODE1);
       break;
     case 2:
       SPI.setDataMode(SPI_MODE2);
       break;
     case 3:
       SPI.setDataMode(SPI_MODE3);
       break;
     default:
       break;
     }
      Serial.write('0');
     break;
   case 0x15:  // SPI Send / Receive
     spi_sendReceive(command);
     break;
   case 0x16:  // SPI Close
     SPI.end();
      Serial.write('0');
     break;


/*****************************************
*****************************************
   ** Servos
*****************************************
*****************************************/
   case 0x17:  // Set Num Servos
     free(servos);
     servos = (Servo*)
malloc(command[2]*sizeof(Servo));
     for(int i=0; i<command[2]; i++)
     {
      servos[i] = Servo();
     }
     if(servos == 0)
     {
      Serial.write('1');
     }
     else
     {
       Serial.write('0');
     }
     break;
   case 0x18:  // Configure Servo
     servos[command[2]].attach(command[3]);
      Serial.write('0');
     break;
   case 0x19:  // Servo Write
     servos[command[2]].write(command[3]);
      Serial.write('0');
     break;
   case 0x1A:  // Servo Read Angle
     Serial.write(servos[command[2]].read());
     break;
   case 0x1B:  // Servo Write uS Pulse
     servos[command[2]].writeMicroseconds(
(command[3] + (command[4]<<8)) );
      Serial.write('0');
     break;
   case 0x1C:  // Servo Read uS Pulse
     retVal =
servos[command[2]].readMicroseconds();
     Serial.write ((retVal & 0xFF));
     Serial.write( (retVal >> 8));
     break;
   case 0x1D:  // Servo Detach
     servos[command[2]].detach();
      Serial.write('0');
     break;

/*****************************************
*****************************************
       **                          LCD

*****************************************
*****************************************/
   case 0x1E:  // LCD Init
      lcd.init(command[2], command[3],
command[4], command[5], command[6],
command[7], command[8], command[9],
command[10], command[11], command[12],
command[13]);
```

```
  Serial.write('0');
  break;
 case 0x1F:  // LCD Set Size
  lcd.begin(command[2], command[3]);
  Serial.write('0');
  break;
 case 0x20:  // LCD Set Cursor Mode
  if(command[2] == 0)
  {
   lcd.noCursor();
  }
  else
  {
   lcd.cursor();
  }
  if(command[3] == 0)
  {
   lcd.noBlink();
  }
  else
  {
   lcd.blink();
  }
  Serial.write('0');
  break;
 case 0x21:  // LCD Clear
  lcd.clear();
  Serial.write('0');
  break;
 case 0x22:  // LCD Set Cursor Position
  lcd.setCursor(command[2], command[3]);
  Serial.write('0');
  break;
 case 0x23:  // LCD Print
  lcd_print(command);
  break;
 case 0x24:  // LCD Display Power
  if(command[2] == 0)
  {
   lcd.noDisplay();
  }
  else
  {
   lcd.display();
  }
  Serial.write('0');
  break;
 case 0x25:  // LCD Scroll
  if(command[2] == 0)
  {
   lcd.scrollDisplayLeft();
  }
  else
  {
   lcd.scrollDisplayRight();
  }
  Serial.write('0');
  break;
 case 0x26: //  LCD Autoscroll
  if(command[2] == 0)
  {
   lcd.noAutoscroll();
  }
  else
  {
   lcd.autoscroll();
  }
  Serial.write('0');
  break;
 case 0x27: // LCD Print Direction
  if(command[2] == 0)
  {
   lcd.rightToLeft();
  }
  else
  {
   lcd.leftToRight();
  }
  Serial.write('0');
  break;
 case 0x28: // LCD Create Custom Char
  for(int i=0; i<8; i++)
  {
   customChar[i] = command[i+3];
  }
  lcd.createChar(command[2], customChar);

  Serial.write('0');
  break;
 case 0x29:  // LCD Print Custom Char
  lcd.write(command[2]);
  Serial.write('0');
  break;



/*****************************************
*****************************************
   ** Continuous Aquisition

*****************************************
*****************************************/
    case 0x2A:  // Continuous Aquisition Mode On
     acqMode=1;
     contAcqPin=command[2];

contAcqSpeed=(command[3])+(command[4]<<8);
     acquisitionPeriod=1/contAcqSpeed;
     iterationsFlt =.08/acquisitionPeriod;
     iterations=(int)iterationsFlt;
     if(iterations<1)
     {
      iterations=1;
     }
     delayTime= acquisitionPeriod;
    if(delayTime<0)
     {
      delayTime=0;
     }
     break;
```

```
     case 0x2B:  // Continuous Aquisition Mode Off
       acqMode=0;
       break;
     case 0x2C:  // Return Firmware Revision
        Serial.write(byte(FIRMWARE_MAJOR));
        Serial.write(byte(FIRMWARE_MINOR));
       break;
     case 0x2D:  // Perform Finite Aquisition
        Serial.write('0');

finiteAcquisition(command[2],(command[3])+(com
mand[4]<<8),command[5]+(command[6]<<8));
       break;

/*****************************************
*****************************************
   ** Stepper

*****************************************
*****************************************/
   #ifdef STEPPER_SUPPORT
     case 0x30:  // Configure Stepper
       if (command[2] == 5){    // Support AFMotor
Shield
         switch (command[3]){
         case 0:
           steppers[command[3]] =
AccelStepper(forwardstep1, backwardstep1);
           break;
         case 1:
           steppers[command[3]] =
AccelStepper(forwardstep2, backwardstep2);
           break;
         default:
           break;
         }
       }
       else if(command[2]==6) {                  // All
other stepper configurations
         steppers[command[3]] = AccelStepper(1,
command[4],command[5],command[6],command[7
]);
       }
       else{
         steppers[command[3]] =
AccelStepper(command[2],
command[4],command[5],command[6],command[7
]);
       }
        Serial.write('0');
       break;
     case 0x31:  // Stepper Write
       AccelStepper_Write(command);
        Serial.write('0');
       break;
     case 0x32:  // Stepper Detach
       steppers[command[2]].disableOutputs();
        Serial.write('0');
       break;
     case 0x33:  // Stepper steps to go
       retVal = 0;
       for(int i=0; i<8; i++){
         retVal += steppers[i].distanceToGo();
       }
       Serial.write( (retVal & 0xFF) );
       Serial.write( (retVal >> 8) );

       break;
   #endif


/*****************************************
*****************************************
   ** Unknown Packet

*****************************************
*****************************************/
    default:      // Default Case
      Serial.flush();
      break;
    }
  }
  else{
    // Checksum Failed, Flush Serial Buffer
    Serial.flush();
  }
}

/*****************************************
*****************************************
**  Functions
*****************************************
*****************************************/

// Writes Values To Digital Port (DIO 0-13).  Pins
Must Be Configured As Outputs Before Being
Written To
void writeDigitalPort(unsigned char command[])
{
  digitalWrite(13, (( command[2] >> 5) & 0x01) );
  digitalWrite(12, (( command[2] >> 4) & 0x01) );
  digitalWrite(11, (( command[2] >> 3) & 0x01) );
  digitalWrite(10, (( command[2] >> 2) & 0x01) );
  digitalWrite(9, (( command[2] >> 1) & 0x01) );
  digitalWrite(8, (command[2] & 0x01) );
  digitalWrite(7, (( command[3] >> 7) & 0x01) );
  digitalWrite(6, (( command[3] >> 6) & 0x01) );
  digitalWrite(5, (( command[3] >> 5) & 0x01) );
  digitalWrite(4, (( command[3] >> 4) & 0x01) );
  digitalWrite(3, (( command[3] >> 3) & 0x01) );
  digitalWrite(2, (( command[3] >> 2) & 0x01) );
  digitalWrite(1, (( command[3] >> 1) & 0x01) );
  digitalWrite(0, (command[3] & 0x01) );
}

// Reads all 6 analog input ports, builds 8 byte
packet, send via RS232.
void analogReadPort()
{
  // Read Each Analog Pin
```

```
  int pin0 = analogRead(0);
  int pin1 = analogRead(1);
  int pin2 = analogRead(2);
  int pin3 = analogRead(3);
  int pin4 = analogRead(4);
  int pin5 = analogRead(5);

  //Build 8-Byte Packet From 60 Bits of Data Read
  char output0 = (pin0 & 0xFF);
  char output1 = ( ((pin1 << 2) & 0xFC) | ( (pin0 >>
8) & 0x03) );
  char output2 = ( ((pin2 << 4) & 0xF0) | ( (pin1 >>
6) & 0x0F) );
  char output3 = ( ((pin3 << 6) & 0xC0) | ( (pin2 >>
4) & 0x3F) );
  char output4 = ( (pin3 >> 2) & 0xFF);
  char output5 = (pin4 & 0xFF);
  char output6 = ( ((pin5 << 2) & 0xFC) | ( (pin4 >>
8) & 0x03) );
  char output7 = ( (pin5 >> 6) & 0x0F );

  // Write Bytes To Serial Port
  Serial.print(output0);
  Serial.print(output1);
  Serial.print(output2);
  Serial.print(output3);
  Serial.print(output4);
  Serial.print(output5);
  Serial.print(output6);
  Serial.print(output7);
}

// Configure digital I/O pins to use for seven
segment display
void sevenSegment_Config(unsigned char
command[])
{
  // Configure pins as outputs and store in
sevenSegmentPins array for use in
sevenSegment_Write
  for(int i=2; i<10; i++)
  {
    pinMode(command[i], OUTPUT);
    sevenSegmentPins[(i-1)] = command[i];
  }
}

//  Write values to sevenSegment display.  Must first
use sevenSegment_Configure
void sevenSegment_Write(unsigned char
command[])
{
  for(int i=1; i<9; i++)
  {
    digitalWrite(sevenSegmentPins[(i-1)],
command[i]);
  }
}

// Set the SPI Clock Divisor
void spi_setClockDivider(unsigned char divider)
{
  switch(divider)
  {
  case 0:
    SPI.setClockDivider(SPI_CLOCK_DIV2);
    break;
  case 1:
    SPI.setClockDivider(SPI_CLOCK_DIV4);
    break;
  case 2:
    SPI.setClockDivider(SPI_CLOCK_DIV8);
    break;
  case 3:
    SPI.setClockDivider(SPI_CLOCK_DIV16);
    break;
  case 4:
    SPI.setClockDivider(SPI_CLOCK_DIV32);
    break;
  case 5:
    SPI.setClockDivider(SPI_CLOCK_DIV64);
    break;
  case 6:
    SPI.setClockDivider(SPI_CLOCK_DIV128);
    break;
  default:
    SPI.setClockDivider(SPI_CLOCK_DIV4);
    break;
  }
}

void spi_sendReceive(unsigned char command[])
{
  if(command[2] == 1)       //Check to see if this is
the first of a series of SPI packets
  {
    spiBytesSent = 0;
    spiCSPin = command[3];
    spiWordSize = command[4];

    // Send First Packet's 8 Data Bytes
    for(int i=0; i<command[5]; i++)
    {
      // If this is the start of a new word toggle CS
LOW
      if( (spiBytesSent == 0) || (spiBytesSent %
spiWordSize == 0) )
      {
        digitalWrite(spiCSPin, LOW);
      }
      // Send SPI Byte
      Serial.print(SPI.transfer(command[i+6]));
      spiBytesSent++;

      // If word is complete set CS High
      if(spiBytesSent % spiWordSize == 0)
      {
        digitalWrite(spiCSPin, HIGH);
      }
    }
```

```
  }
  else
  {
   // SPI Data Packet - Send SPI Bytes
   for(int i=0; i<command[3]; i++)
   {
    // If this is the start of a new word toggle CS
LOW
    if( (spiBytesSent == 0) || (spiBytesSent %
spiWordSize == 0) )
    {
     digitalWrite(spiCSPin, LOW);
    }
    // Send SPI Byte
    Serial.write(SPI.transfer(command[i+4]));
    spiBytesSent++;

    // If word is complete set CS High
    if(spiBytesSent % spiWordSize == 0)
    {
     digitalWrite(spiCSPin, HIGH);
    }
   }
  }
}

// Synchronizes with LabVIEW and sends info about
the board and firmware (Unimplemented)
void syncLV()
{
  Serial.begin(DEFAULTBAUDRATE);
  i2cReadTimeouts = 0;
  spiBytesSent = 0;
  spiBytesToSend = 0;
  Serial.flush();
}

// Compute Packet Checksum
unsigned char checksum_Compute(unsigned char
command[])
{
  unsigned char checksum;
  for (int i=0; i<(COMMANDLENGTH-1); i++)
  {
   checksum += command[i];
  }
  return checksum;
}

// Compute Packet Checksum And Test Against
Included Checksum
int checksum_Test(unsigned char command[])
{
  unsigned char checksum =
checksum_Compute(command);
  if(checksum == command[COMMANDLENGTH-
1])
  {
   return 0;
  }
  else
  {
   return 1;
  }
}

// Stepper Functions
#ifdef STEPPER_SUPPORT
  void AccelStepper_Write(unsigned char
command[]){
   int steps = 0;
   int step_speed = 0;
   int acceleration = 0;

   //Number of steps & speed are a 16 bit values,
split for data transfer. Reassemble 2 bytes to an int
16
   steps = (int)(command[5] << 8) + command[6];
   step_speed = (int)(command[2] << 8) +
command[3];
   acceleration = (int)(command[7] << 8) +
command[8];

   steppers[command[4]].setMaxSpeed(step_speed);

   if (acceleration == 0){
    //Workaround AccelStepper bug that requires
negative speed for negative step direction
     if (steps < 0) step_speed = -step_speed;
     steppers[command[4]].setSpeed(step_speed);
     steppers[command[4]].move(steps);
   }
   else {

steppers[command[4]].setAcceleration(acceleration)
;
     steppers[command[4]].move(steps);
   }
  }
#endif


void sampleContinously()
{

  for(int i=0; i<iterations; i++)
  {
    retVal = analogRead(contAcqPin);
    if(contAcqSpeed>1000) //delay Microseconds is
only accurate for values less that 16383
    {
     Serial.write( (retVal >> 2));
     delayMicroseconds(delayTime*1000000);
//Delay for neccesary amount of time to achieve
desired sample rate
    }
    else
    {
      Serial.write( (retVal & 0xFF) );
      Serial.write( (retVal >> 8));
```

```
      delay(delayTime*1000);
    }
  }
}
void finiteAcquisition(int analogPin, float
acquisitionSpeed, int numberOfSamples)
{
  //want to exit this loop every 8ms
   acquisitionPeriod=1/acquisitionSpeed;

  for(int i=0; i<numberOfSamples; i++)
  {
    retVal = analogRead(analogPin);

    if(acquisitionSpeed>1000)
    {
      Serial.write( (retVal >> 2));

delayMicroseconds(acquisitionPeriod*1000000);
    }
    else
    {
      Serial.write( (retVal & 0xFF) );
      Serial.write( (retVal >> 8));
      delay(acquisitionPeriod*1000);
    }
  }
}

void lcd_print(unsigned char command[])
{
  if(command[2] != 0)
  {
    // Base Specified By User
    int base = 0;
    switch(command[2])
    {
      case 0x01:  // BIN
        base = BIN;
        break;
      case 0x02:  // DEC
        base = DEC;
        break;
      case 0x03:  // OCT
        base = OCT;
        break;
      case 0x04:  // HEX
        base = HEX;
        break;
      default:
        break;
    }
    for(int i=0; i<command[3]; i++)
    {
      lcd.print(command[i+4], base);
    }
  }
  else
  {
    for(int i=0; i<command[3]; i++)
    {
      lcd.print((char)command[i+4]);

    }
  }
  Serial.write('0');
}
```