```c
#include <msp430.h>


#define bufferSize 150              // Buffer size for UART receiving
#define numPhases 8                 // Defined for easier switching between full and half steps during debugging
#define xControlRefresh 0x1388      // Delay for x Control Loop during control operation (20ms)
#define xRegularRefresh 0xC350      // Delay for x Control Loop during non-control operation (200ms)
// Note: For smoother slider DC control change xRegularRefresh to 0x1388 (control tuning is off when this happens though)

// Control Constants
#define Kd 0xFFFF/123
#define Kdy 0xFFFF/123*2*20/400     // Conversion from Input Y Val to half steps
#define Kenc (4*40)/(20.4*48)       // Conversion from
#define tau 0.02375                 // Time Constant solved in rise time calculations
#define Kp 1/tau*0.23               // Proportional Controller using theoretical 1/tau * a tuneable value
#define Ktim xRegularRefresh/0xFFFF // Conversion from input speed to prop. control refresh (reset timer)


// UART Variables
unsigned volatile int circBuffer[bufferSize];                   // For storing received data packets
unsigned volatile int head = 0;                                 // circBuffer head
unsigned volatile int tail = 0;                                 // circBuffer tail
unsigned volatile int length = 0;                               // circBuffer length
unsigned volatile char* bufferFullMsg = "Buffer is full";       // Message to print when buffer is full
unsigned volatile char* bufferEmptyMsg = "Buffer is empty";     // Message to print when buffer is empty
unsigned volatile int rxByte = 0;                               // Temporary variable for storing each received byte
volatile int rxFlag = 0;                                        // Received data flag, triggered when a packet is received
volatile int rxIndex = 0;                                       // Counts bytes in data packet

// Stepper Control Variables
unsigned int halfStepLookupTable[numPhases][4] =                // Phases for stepping
```

```c
{
 // Full Step
// {1, 0, 0, 0},
// {0, 0, 1, 0},
// {0, 1, 0, 0},
// {0, 0, 0, 1}
 // Half Step
 {1, 0, 0, 0},
 {1, 0, 1, 0},
 {0, 0, 1, 0},
 {0, 1, 1, 0},
 {0, 1, 0, 0},
 {0, 1, 0, 1},
 {0, 0, 0, 1},
 {1, 0, 0, 1}
};
volatile int contStepperMode = 0;                                   // 0 = No power to motor, 1 = CW dir continuous, -1 = CCW dir continuous, 2 = single step mode

// Variables for X Control (DC)
unsigned int currentTA0, currentTA1;
unsigned volatile int xr = 0;                                       // Goal loc for X controller
unsigned volatile int xControlFlag = 0;                             // Signals whether X is in a control loop
volatile int error = 0;                                             // Error between encoder and X goal
const double errorMult = (Kd)*(Kenc);                               // Multiplier for scaling of encoder count
const double errorTimerMult = Kp * xControlRefresh/0xFFFF;          // Multiplier for prop controller scaled to control loop delay
volatile unsigned int encoderCount = 0;                             // X Location based on encoder

// Variables for Y Control (Stepper)
unsigned volatile int yr = 0;                                       // Goal loc for Y controller
unsigned volatile int yControlFlag = 0;                             // Signals whether Y is in a control loop
unsigned int yLoc = 0;                                              // Current location of Y

// Variables for XY Control
volatile double xStep = 0;                                          // Size of X step to match Y steps in given loc
const double locErrorTolerance = 0.22*Kd;                           // Tolerance for where to stop control loop for X
volatile unsigned int xGoal = 0;                                    // Final goal
```

```
  of X axis

// Function to update the stepper coil voltages based on lookup table and current position in cycle
void updateStepperCoils(){
    if (halfStepLookupTable[yLoc%numPhases][0] == 1)
        P1OUT |= BIT4;
    else
        P1OUT &= ~BIT4;
    if (halfStepLookupTable[yLoc%numPhases][1] == 1)
        P1OUT |= BIT5;
    else
        P1OUT &= ~BIT5;
    if (halfStepLookupTable[yLoc%numPhases][2] == 1)
        P3OUT |= BIT4;
    else
        P3OUT &= ~BIT4;
    if (halfStepLookupTable[yLoc%numPhases][3] == 1)
        P3OUT |= BIT5;
    else
        P3OUT &= ~BIT5;
}

// Function to transmit a UART package given arguments for package
void transmitPackage(unsigned int instrByte, unsigned int dataByte1, unsigned int dataByte2){
    unsigned int decoderByte = 0;
    if (dataByte1 == 255){
        decoderByte |= 2;
        dataByte1 = 0;
    }
    if (dataByte2 == 255){
        decoderByte |= 1;
        dataByte2 = 0;
    }
    while (!(UCA1IFG & UCTXIFG));
    UCA1TXBUF = 255;
    while (!(UCA1IFG & UCTXIFG));
    UCA1TXBUF = instrByte;
    while (!(UCA1IFG & UCTXIFG));
    UCA1TXBUF = dataByte1;
    while (!(UCA1IFG & UCTXIFG));
    UCA1TXBUF = dataByte2;
    while (!(UCA1IFG & UCTXIFG));
    UCA1TXBUF = decoderByte;
    while (!(UCA1IFG & UCTXIFG));
}

// Main Loop for initialization, reading of UART buffer, and starting commands
```

```c
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;    // stop watchdog timer

    // Configure Clocks
    CSCTL0 = 0xA500;                             // Write password to modify CS registers
    CSCTL1 = DCORSEL;                            // DCO = 16 MHz
    CSCTL2 |= SELM_3 + SELS_3 + SELA_3;          // MCLK = DCO, ACLK = DCO, SMCLK = DCO
    CSCTL3 |= DIVS_5;                            // Set divider for SMCLK (/32) -> SMCLK 500kHz

    // Configure timer B2 for DC Motor
    TB2CTL |= TBSSEL_2 + MC_1 + ID_1 + TBIE;     // SCLK, up mode, div by 2, overflow interrupt enable
    TB2CCTL1 |= OUTMOD_7;                        // CCR1 reset/set
    TB2CCR0 = xRegularRefresh;                   // CCR0: control loop delay time
    TB2CCR1 = 0x9C40 * Ktim;                     // CCR1 PWM duty cycle: DC Motor PWM rate (scaled based on control loop delay time)

    // Configure timer B0 for Stepper Motor
    TB0CTL |= TBSSEL_1 + MC_1;                   // ACLK, up mode (16MHz)
    TB0CCTL0 |= CCIE;                            // CCR0 interrupt enable
    TB0CCR0 = 0xFFFF;                            // CCR0: interrupt for half step phase switching

    // Configure Timers for DC Encoder
    // Timer A0 for DWN Encoder
    TA0CTL |= TASSEL_0 + MC_1 + TACLR;           // Input pin clock, up mode, clear timer val
    TA0CCR0 = 0xFFFF;
    // Timer A1 for UP Encoder
    TA1CTL |= TASSEL_0 + MC_1 + TACLR;           // Input pin clock, up mode, clear timer val
    TA1CCR0 = 0xFFFF;

    // Configure Timer B1 for Duty Cycle of Stepper Pins
    TB1CTL |= TBSSEL_1 + MC_1 + ID_1;            // ACLK, Up mode, Div by 2 -> 8MHz
    TB1CCTL0 = CCIE;                             // CCR0: Enabling Stepper Phases
    TB1CCTL1 = CCIE;                             // CCR1: Turning off stepper phases
    TB1CCR0 = 0x3E8;                             // 0.125ms full pwm period
    TB1CCR1 = 0x11C;                             // 28.4% duty cycle

    // Configure outputs for DC PWM Pin
```

```c
    P2SEL0 |= BIT1;
    P2DIR |= BIT1;

    // Configure outputs for DC AIN1 and AIN2 Pins
    P3DIR |= BIT6 + BIT7;                    // Output pins for AIN2 and AIN1 respectively

    // Configure outputs for Stepper A1 A2 B1 B2 Pins
    P1DIR |= BIT4 + BIT5;                    // AIN2 and AIN1 Pins respectively
    P3DIR |= BIT4 + BIT5;                    // BIN2 and BIN1 Pins respectively

    // Setup Pins for DC Encoder Interrupt Capture
    P1SEL1 |= BIT1 + BIT2;

    // Configure ports for UART
    P2SEL0 &= ~(BIT5 + BIT6);
    P2SEL1 |= BIT5 + BIT6;

    // Configure UART
    UCA1CTLW0 |= UCSSEL0;
    UCA1MCTLW = UCOS16 + UCBRF0 + 0x4900;    // Define UART as 19200baud rate
    UCA1BRW = 52;
    UCA1CTLW0 &= ~UCSWRST;
    UCA1IE |= UCRXIE;                        //enable UART receive interrupt
    _EINT();                                 //Global interrupt enable

    // Circular Buffer Data Processing Variables
    unsigned volatile int commandByte, dataByte1, dataByte2, escapeByte, dataByte;

    while (1)
    {
        if (rxFlag)
        {
            // Get escape byte and command byte from buffer
            escapeByte = circBuffer[head - 1];
            commandByte = circBuffer[head - 4];

            // Handle the Data Bytes
            // Check if the first bit of escape byte is 1 and if so set dataByte2 to 255
            if (escapeByte & 1) { dataByte2 = 255; }
            // Else, dataByte2 gets the value from the buffer
            else { dataByte2 = circBuffer[head - 2]; }
            // Check if the second bit of escape byte is 1 and if so set dataByte1 to 255
            if (escapeByte & 2) { dataByte1 = 255; }
            // Else, dataByte1 gets the value from the buffer
            else { dataByte1 = circBuffer[head - 3]; }
```

```c

            // DataByte gets the combination of dataByte1 & dataByte2
            dataByte = (dataByte1 << 8) + dataByte2;


            // Handle the command Bytes
            switch(commandByte)
            {
            case 0: // Stop DC Motor
                P3OUT &= ~(BIT6 + BIT7);
                xControlFlag = 0;
                TB2CCR1 = 0;
                break;
            case 1: // CW DC Motor
                P3OUT |= BIT7;
                P3OUT &= ~BIT6;
                TB2CCR1 = dataByte;                  // PWM for DC
                break;
            case 2: // CCW DC Motor
                P3OUT |= BIT6;
                P3OUT &= ~BIT7;
                TB2CCR1 = dataByte;                  // PWM for DC
                break;
            case 3: // Single Step CW
                contStepperMode = 2;                 // Single step mode
                yLoc++;                              // Responded to by Timer B1 Interrupts
                break;
            case 4: // Single Step CCW
                contStepperMode = 2;                 // Single step mode
                yLoc--;                              // Responded to by Timer B1 Interrupts
                break;
            case 5: // Continuous Step CW
                contStepperMode = 1;                 // Continuous step mode pos
                TB0CCR0 = 0xFFFF - dataByte;         // PWM sub so that large input = fast speed
                break;
            case 6: // Continuous Step CCW
                contStepperMode = -1;                // Continuous step mode neg
                TB0CCR0 = 0xFFFF - dataByte;         // PWM sub so that large input = fast speed
                break;
            case 7: // Stop Stepper Continuous
                contStepperMode = 0;                 // Cuts power to stepper phases
                break;
            case 8: // Zero the Encoder
                TA0R = 0;                            // Zero all encoder
```

```
                tracking variables
                TA1R = 0;
                currentTA0 = 0;
                currentTA1 = 0;
                break;
            case 9: // Go To X Loc
                xr = dataByte;                      // Gives goal for X to reach
                TB2CCR0 = xControlRefresh;          // Changes X control loop to faster delay
                xControlFlag = 1;                   // Enables X control
                break;
            case 10: // Zero The Stepper
                contStepperMode = 2;                // Changes to single step mode
                while(yLoc%8 != 1){                 // Steps until current step is in 0 position of lookup table
                    yLoc++;
                }
                yLoc = 0;                           // Sets y location to 0
                break;
            case 11: // Go To Y Loc
                yr = dataByte/(Kdy);                // Scale input (0x0-0xFFFF) to # of half step steps
                yControlFlag = 1;                   // Enable y control loop
                TB0CCR0 = 0xFFFF - 0xBD4C;          // Set default Y speed
                if (yLoc < yr){                     // Sets direction of stepper rotation (dealt with in control loop after this)
                    contStepperMode = 1;
                }
                else if (yLoc > yr){
                    contStepperMode = -1;
                }
                break;
            case 12: // Send Y Loc for XY Movement  // Sent first when changing X and Y in straight line
                contStepperMode = 0;                // Stops stepper power
                xControlFlag = 0;                   // Clears all control flags and vars to wait for rest of commands
                yControlFlag = 0;
                xStep = 0;
                yr = dataByte/(Kdy);                // Sets Y step goal
//              transmitPackage(1, yr>>8, yr & 0xFF); //Transmits debugging value of y # of steps
                break;
            case 13: // Send X Loc for XY Movement // Sent second when changing X and Y in straight line
                xGoal = dataByte;                   // Sets end goal of X position
```

```c
                int yStep = abs(yr - yLoc);         // Calculates overall Y steps
                xStep = (dataByte - encoderCount*Kd*Kenc);  // Calculates size of x step if travel will happen in a single jump
                if (yStep != 0){                    // Scales X step by number of y steps if y needs to move
                    xStep = xStep/yStep;
                }
                else {                              // Scales x to take 600 steps as default step size if Y doesnt need to move
                    xStep = xStep/600;
                }
//                  transmitPackage(2, (int)xStep>>8, (int)xStep & 0xFF); // Trasmits debugging value of x step size
                break;
            case 14: // Send Speed for XY Movement and start // Final command sent for XY move in straight line
                TB0CCR0 = (0xFFFF - dataByte);      // Sets speed of travel (time between steps)
                TB2CCR0 = xControlRefresh;          // Change X control loop to faster delay
                if (xStep != 0){                    // Turns on X control only if X is changing
                    xControlFlag = 1;
                }
                yControlFlag = 1;                   // Turns on Y control
                if (yLoc < yr){                     // Sets Y direction
                    contStepperMode = 1;
                }
                else if (yLoc > yr){
                    contStepperMode = -1;
                }
                break;
            default: // No known command
                break;
            }

//          Remove the processed bytes from the buffer
            length -= 5;                        // Decrease length by 5
            if (bufferSize - tail <= 5) { tail = 0; }    // Check if tail at end of buffer and if so put it at start
            else { tail += 5; }                 // Else, increase tail by 5

            // reset the data received flag
            rxFlag = 0;
        }
    }
    return 0;
}
```

```c

// UART interrupt to fill receive buffer with data sent from C# program
#pragma vector = USCI_A1_VECTOR
__interrupt void USCI_A1_ISR(void)
{
    rxByte = UCA1RXBUF;                 // rxByte gets the received byte

    // Check if 255 was received
    if (rxByte == 255 || rxIndex > 0)
    {
        // Check that the buffer isn't full
        if (length < bufferSize)
        {
            circBuffer[head] = rxByte;      // Buffer gets received byte at head
            length++;                       // Increment length

            if (head == bufferSize) { head = 0; }   // Check if head at end of buffer and if so put it at start
            else { head++; }                // Else, increment head

            // Check if receiving index is 4 or greater and if so reset
            if (rxIndex >= 4)
            {
                rxIndex = 0;                // Reset receiving index
                rxFlag = 1;                 // Set the data received flag
            }
            else { rxIndex++; }             // Increment rxIndex
        }
    }
}

// Timer B0 CCR0 Interrupt: Y Control loop (updates X step by step during XY control mode
#pragma vector = TIMER0_B0_VECTOR
__interrupt void TriggerTimer (void){
    // During XY control mode increments by xStep
    if (xControlFlag && yControlFlag){
        xr = xr + xStep;
    }

    // If continuous stepper mode increase or decrease yLoc accordingly
    if (contStepperMode == 1){
        yLoc++;
    }
    else if (contStepperMode == -1){
        yLoc--;
    }

```

```c
    // If Y is at goal and in control mode
    if (yControlFlag == 1 && yLoc == yr){
        yControlFlag = 0;                       // Turn off Y control mode
        xr = xGoal;                             // Set X to go to final goal (compensates for step rounding issue)
        contStepperMode = 0;                    // Stops continuous stepper mode
    }

    TB0CCTL0 &= ~CCIFG;                         // Reset interrupt flag
}

// Timer B2 CCR1 Interrupt: Updates x Position and handles X Control Loop
#pragma vector = TIMER2_B1_VECTOR
__interrupt void SendEncoderCount(void){
    // Reads current encoder position
    unsigned int instructionByte = 0;               // Set instruction byte for loop refresh non-control speed
    TA0CTL &= MC_0;                                 // Turn off timers to read register (unstable if still on)
    TA1CTL &= MC_0;
    currentTA0 = TA0R;                              // Read current timer counts
    currentTA1 = TA1R;
    TA0CTL |= MC_1;                                 // Turn timers back on
    TA1CTL |= MC_1;
    encoderCount = currentTA0 - currentTA1;         // Update encoder count UpCount - DownCount
    if (currentTA1 > currentTA0){                   // Sets encoder count to 0 if negative (overflow)
        encoderCount = 0;
    }

    // Do Controls for DC Motor
    if (xControlFlag == 1){
        instructionByte = 1;                        // Set instruction byte for return message signifying in control loop delay time
        error = xr - (encoderCount*errorMult);      // Calculate error between current x goal (not final) and scaled encoder count
        TB2CCR1 = abs(error)*errorTimerMult;        // Change speed according to error and proportional controller
        if (error > locErrorTolerance){             // Choose direction based on if current location is past or before goal (by tolerance)
            P3OUT |= BIT7;
            P3OUT &= ~BIT6;
        }
        else if (error < -locErrorTolerance){
            P3OUT |= BIT6;
            P3OUT &= ~BIT7;
        }
        else if (yControlFlag == 0){                // If error is within tolerance
```

```c
            and no XY control exit X control
                P3OUT &= ~(BIT6 + BIT7);            // Stop DC motor
                TB2CCR0 = xRegularRefresh;          // Go back to control loop regular time delay
                xControlFlag = 0;                   // Turn off X control
            }
        }
    transmitPackage(instructionByte, encoderCount>>8, encoderCount & 0xFF);     // Transmit the current encoder value for C# program to track
    TB2CTL &= ~TBIFG;                               // Reset interrupt flag
}

// Timer B1 CCR1 Interrupt: Turn off stepper phases for PWM
#pragma vector = TIMER1_B1_VECTOR
__interrupt void TurnOffStepperPhases(void){
    P1OUT &= ~(BIT4 + BIT5);
    P3OUT &= ~(BIT4 + BIT5);
    TB1CCTL1 &= ~CCIFG;
}

// Timer B1 CCR0 Interrupt: Turn on proper stepper phases for PWM
#pragma vector = TIMER1_B0_VECTOR
__interrupt void TurnOnStepperPhases(void){
    if (contStepperMode != 0){
        updateStepperCoils();
    }
    TB1CCTL0 &= ~CCIFG;
}
```