```c
/*
 * FileName: main.c
 * Version: 1
 *
 * Created: 11/11/2022 8:46:12 AM
 * Author: Ethan Zeronik
 *
 * Operations: crane project source code
 *
 * Hardware:
 *   Atmega2560          micro controller
 *   Port L              LED reposnse bar
 *   Port C.4            stepper motor limit switch
 *   Port C.0-3          stepper motor
 *   Port A.0-3          button inputs
 *   Port B.5-6          servo outputs
 */

/* NOTE: Includes */
// standard include for the atmega program
#include <avr/io.h>
#include <avr/interrupt.h>

#include <stdio.h>

// adds stepper motor module (1/6)
#include "StepperMotor.h"
// adds adc for input potentiometers (2/6)
#include "AnalogToDigital.h"
// adds pwm for servo motors (3/6)
#include "CraneServo.h"
// adds UART and Bluetooth for communication (4/6)
#include "CraneCommunication.h"
// adds EEPROM read/write for persistant data storage (5/6)
#include "CraneEeprom.h"
// adds custom delay functions (6/6)
#include "CraneDelay.h"

/* NOTE: Types and Structs */
// type to hold data for serial connections
typedef struct connectionBuffer_t
{
    // the buffer to store the data in
    char    buffer[128];
    // the current index of the last set byte in the buffer
    uint8_t index;
    // whether or not the buffer is done being written to
    uint8_t readFlag;
} connectionBuffer_t;
// type to hold the position of the servos and motor
typedef struct cranePosition_t
{
    // motor position in steps
    int16_t motorTicks;
    // arm position in 0-255 ticks
    uint8_t armTicks;
    // plunger position in 0-255 ticks
    uint8_t plungerTicks;
} cranePosition_t;

/* NOTE: Custom Macros */
// the three main states of the application
#define calibrateState 0
#define actionState    1
#define recordState    2

// command definition
#define recordModeCommand    "Calibrate"
#define calibrateModeCommand "Reset"
#define runCommand           "Run"
#define recordCommand        "Record"
#define getCommand           "Get"

// custom buttons & switches
#define limitSwitch     (PINC & 0x10)
#define leftButton      (PINA & 0x01)
#define rightButton     (PINA & 0x02)
#define recordButton    (PINA & 0x04)
#define calibrateButton (PINA & 0x08)

// custom led bar
#define stateLed PORTL

// custom servos ids
#define armServo     0
#define plungerServo 1

// defined values for where default spots of the servo are
#define armStartPosition     150
#define plungerStartPosition 230

// record information
#define recordLength  6
#define eepromAddress 0x0555

/* NOTE: Global Variables */
// the state of the application
uint8_t         applicationState = calibrateState;
// the state of all the moving parts of the crane
cranePosition_t craneState       = {
          .motorTicks   = 0,
          .armTicks     = 0,
          .plungerTicks = 0,
};
// the EEPROM recorded moves to make
cranePosition_t             recordedMoves[recordLength];
// the moves to save to eemprom
cranePosition_t             currentMoves[recordLength];
// current spot in the instance moves
uint8_t                     currentMoveIndex = 0;
// the input buffer for serial
volatile connectionBuffer_t serialInputData  = {
     .buffer   = {0},
     .index    = 0,
     .readFlag = 0,
};
// the input buffer for bluetooth
volatile connectionBuffer_t bluetoothInputData = {
    .buffer   = {0},
    .index    = 0,
    .readFlag = 0,
};

/* NOTE: Function prototypes */
// inits IO ports
void    IO_init(void);
// gets the saved values from the eeprom
void    CRANE_getMovesFromEeprom(uint16_t addr);
// saves the current recorded moves to eeprom
void    CRANE_saveMovesToEeprom(uint16_t addr);
// takes a string and checks if the buffer matches the value exactly
// strictmode resets readflag
// returns 1 if true, else 0
uint8_t doesBufferMatch(volatile connectionBuffer_t buf, uint8_t strictMode, char const * const pStr);
// compares to strings
// if the match it returns 1, else 0
uint8_t stringCompare(char const * const pStrOne, char const * const pStrTwo);

/* NOTE: Application implementation */
// the main loop of the function, provided to us
int main(void)
{
    // start the gpio
    IO_init();

    // init the ADC and the servo control
    ADC_init();
    CRANE_initServos();
    CRANE_startServos();

    // in this case, 1 is CCW, and 0 is CW
    // pass in the port and register we want to use for the motor
    SM_init(&DDRC, &PORTC);

    // get the stored recorded data from the EEPROM
    CRANE_getMovesFromEeprom(eepromAddress);

    // turn on the serial on port 0 at 9600 baud
    CRANE_initSerial(9600);
    CRANE_sendSerial("Crane online\r\n");

    // turn on the blutooth connection
    CRANE_initBluetooth(9600);
    CRANE_sendBluetooth("Crane online\r\n");

    // start the delay timer
    CRANE_initTimer();
```

```c

    // start global interrupts
    sei();

    while(1)
    {
        if(bluetoothInputData.readFlag)
        {
            CRANE_sendSerial(bluetoothInputData.buffer);
        }

        // DEBUG: get the state
        // if we get the command for record mode and we are not already in record mode
        if((doesBufferMatch(serialInputData, 0, recordModeCommand) && (applicationState != recordState)) || (doesBufferMatch(bluetoothInputData, 0, recordModeCommand) && (applicationState != recordState)) || (recordButton && (applicationState != recordState)))
        {
            CRANE_sendSerial("Entering record mode...\r\n");
            CRANE_sendBluetooth("Entering record mode...\r\n");

            applicationState            = recordState;
            serialInputData.readFlag    = 0;
            bluetoothInputData.readFlag = 0;

            while(recordButton)
            {
                // do nothing until we let go
            }
        }
        // check for reset command
        else if((doesBufferMatch(serialInputData, 0, calibrateModeCommand) && (applicationState != calibrateState)) || (doesBufferMatch(bluetoothInputData, 1, calibrateModeCommand) && (applicationState != calibrateState)) || (calibrateButton && (applicationState != calibrateState)))
        {
            CRANE_sendSerial("Resetting...\r\n");
            CRANE_sendBluetooth("Resetting...\r\n");

            applicationState            = calibrateState;
            serialInputData.readFlag    = 0;
            bluetoothInputData.readFlag = 0;

            while(calibrateButton)
            {
                // do nothing until we let go
            }
        }

        // display state on leds
        stateLed = (stateLed & 0xfc) | applicationState;

        // DEBUG: check for the get command
        if(doesBufferMatch(serialInputData, 0, getCommand) || doesBufferMatch(bluetoothInputData, 0, getCommand))
        {
            char response[96];
            sprintf(response, "Motor is at %i, Plunger is at %u, Arm is at %u\r\n", craneState.motorTicks, craneState.plungerTicks, craneState.armTicks);

            serialInputData.readFlag    = 0;
            bluetoothInputData.readFlag = 0;

            CRANE_sendSerial(response);
            CRANE_sendBluetooth(response);
        }

        // main application switch case
        switch(applicationState)
        {
            // DEBUG: the action case
            case actionState:
            {
                if(leftButton || doesBufferMatch(serialInputData, 0, runCommand) || doesBufferMatch(bluetoothInputData, 0, runCommand))
                {
                    // HACK: potential multiple select
                    serialInputData.readFlag    = 0;
                    bluetoothInputData.readFlag = 0;

                    for(uint8_t i = 0; i < recordLength; i++)
                    {
                        char response[32];
                        sprintf(response, "Running recorded step %u...\r\n", i + 1);

                        CRANE_sendSerial(response);
                        CRANE_sendBluetooth(response);
                        stateLed = (stateLed & 0x03) | 1 << (i + 2);

                        // calculate the relative movement of the arm
                        int16_t moveSteps = recordedMoves[i].motorTicks - craneState.motorTicks;

                        // move motor
                        if(moveSteps > 0)
                        {
                            SM_moveStepsSigned(stepperModeHalf, 0, moveSteps);
                        }
                        else if(moveSteps < 0)
                        {
                            SM_moveStepsSigned(stepperModeHalf, 1, -1 * moveSteps);
                        }

                        // lerp!
                        for(uint8_t j = 1; j < 101; j++)
                        {
                            CRANE_setServoPosition(armServo, craneState.armTicks + ((recordedMoves[i].armTicks - craneState.armTicks) * ((float)j / 100)));
                            CRANE_delayMs(5);
                        }

                        for(uint8_t j = 1; j < 101; j++)
                        {
                            CRANE_setServoPosition(plungerServo, craneState.plungerTicks + ((recordedMoves[i].plungerTicks - craneState.plungerTicks) * ((float)j / 100)));
                            CRANE_delayMs(5);
                        }

                        // set our state
                        craneState.motorTicks   = recordedMoves[i].motorTicks;
                        craneState.armTicks     = recordedMoves[i].armTicks;
                        craneState.plungerTicks = recordedMoves[i].plungerTicks;

                        CRANE_delayMs(100);
                    }

                    // reset state led
                    stateLed = (stateLed & 0x03);

                    while(leftButton)
                    {
                        // do nothing until we let go
                    }
                }
            }
            break;

            // DEBUG: the record case
            case recordState:
            {
                uint8_t armPosition     = 255 * ADC_getTenBitValue(0);
                uint8_t plungerPosition = 255 * ADC_getTenBitValue(1);
                int16_t moveSteps       = 8;

                // display the step we are recording
                stateLed = (stateLed & 0x03) | 1 << (currentMoveIndex + 2);

                // manually move the motor
                if(rightButton)
                {
                    SM_moveStepsSigned(stepperModeHalf, 0, moveSteps);
                    CRANE_delayMs(10);
                }
                else if(leftButton)
                {
                    SM_moveStepsSigned(stepperModeHalf, 1, moveSteps);
                    CRANE_delayMs(10);

                    // negate for current position
                    moveSteps *= -1;
                }
                else
                {
                    moveSteps = 0;
                }

                // manually move the servos
                CRANE_setServoPosition(armServo, armPosition);
                CRANE_setServoPosition(plungerServo, plungerPosition);

                // update the current positions
                craneState.motorTicks += moveSteps;
```

```c
                craneState.armTicks     = armPosition;
                craneState.plungerTicks = plungerPosition;

                // if we press the record button save the position
                if(recordButton || doesBufferMatch(serialInputData, 0, recordCommand) || doesBufferMatch(bluetoothInputData, 0, recordCommand))
                {
                    serialInputData.readFlag    = 0;
                    bluetoothInputData.readFlag = 0;

                    char response[64];
                    sprintf(response, "Recording step %u out of 6...\r\n", currentMoveIndex + 1);

                    CRANE_sendSerial(response);
                    CRANE_sendBluetooth(response);

                    sprintf(response, "Recorded {%i,%u,%u}...\r\n", craneState.motorTicks, craneState.plungerTicks, craneState.armTicks);

                    CRANE_sendSerial(response);
                    CRANE_sendBluetooth(response);

                    // reset state led
                    stateLed = (stateLed & 0x03);

                    if(currentMoveIndex < recordLength)
                    {
                        currentMoves[currentMoveIndex++] = craneState;
                    }

                    if(currentMoveIndex >= recordLength)
                    {
                        CRANE_saveMovesToEeprom(eepromAddress);
                        currentMoveIndex = 0;

                        // done recording, back to action state after zeroing
                        // home ---> action(play)
                        applicationState = calibrateState;
                    }

                    while(recordButton)
                    {
                        // do nothing until we let go
                    }
                }
            }
            break;

            // DEBUG: the default case will be the home case
            case calibrateState:
            default:
            {
                // 0 is up 255 is down
                CRANE_setServoPosition(armServo, armStartPosition);
                // 255 is close 0 is open
                CRANE_setServoPosition(plungerServo, plungerStartPosition);

                // move CW for one second to ensure the limit switch is not set
                SM_moveTime(stepperModeHalf, 0, 1000, 3);

                // while not hitting the switch
                while(!limitSwitch)
                {
                    // then move CCW a bit at the time until we hit the limit switch
                    SM_moveStepsSigned(stepperModeHalf, 1, 24);
                }

                // then move 30 degrees back to center the arm
                SM_movePosition(stepperModeHalf, 35);

                // set the current position
                craneState.motorTicks   = 0;
                craneState.armTicks     = armStartPosition;
                craneState.plungerTicks = plungerStartPosition;

                // then set it to action state
                applicationState = actionState;
            }
            break;
        }
    }
}

// interrupt handling for the serial connection
ISR(serialInterrupt)
{
    if(serialData != '\r' && serialData != '\n' && serialData != '\0' && (serialInputData.index < 127))
    {
        // add to array
        serialInputData.buffer[serialInputData.index]     = serialData;
        serialInputData.buffer[serialInputData.index + 1] = '\0';

        serialInputData.index++;
    }
    else
    {
        // set update flag
        serialInputData.readFlag = 1;
        // reset message index
        serialInputData.index    = 0;
    }
}

// interrupt handling for the bluetooth connection
ISR(bluetoothInterrupt)
{
    if(bluetoothData != '\r' && bluetoothData != '\n' && bluetoothData != '\0' && (bluetoothInputData.index < 127))
    {
        // add to array
        bluetoothInputData.buffer[bluetoothInputData.index]     = bluetoothData;
        bluetoothInputData.buffer[bluetoothInputData.index + 1] = '\0';

        bluetoothInputData.index++;
    }
    else
    {
        // set update flag
        bluetoothInputData.readFlag = 1;
        // reset message index
        bluetoothInputData.index    = 0;
    }
}

/* NOTE: Function implementations */
void IO_init(void)
{
    // port c.4 is the limit switch
    DDRC  = 0x00;
    PORTC = 0x10;

    // port a.0-1 are for left and right
    DDRA  = 0x00;
    PORTA = 0xff;

    DDRL  = 0xff;
    PORTL = 0x00;
}

void CRANE_getMovesFromEeprom(uint16_t addr)
{
    uint16_t address = addr;

    for(uint8_t i = 0; i < recordLength; i++)
    {
        uint16_t motorTicks = 0;

        // read the motor position
        motorTicks = CRANE_eepromReadChar(address++) << 8;
        motorTicks += CRANE_eepromReadChar(address++);

        recordedMoves[i].motorTicks = motorTicks;

        // read the arm position
        recordedMoves[i].armTicks = CRANE_eepromReadChar(address++);

        // read the plunger position
        recordedMoves[i].plungerTicks = CRANE_eepromReadChar(address++);
    }
}

void CRANE_saveMovesToEeprom(uint16_t addr)
{
    uint16_t address = addr;

    for(uint8_t i = 0; i < recordLength; i++)
    {
        // cheat by directly moving into our recorded array
```

```c
        recordedMoves[i] = currentMoves[i];

        // save the motor position
        CRANE_eepromWriteChar((currentMoves[i].motorTicks & 0xff00) >> 8, address++);
        CRANE_eepromWriteChar((currentMoves[i].motorTicks & 0x00ff), address++);

        // save the arm position
        CRANE_eepromWriteChar(currentMoves[i].armTicks, address++);

        // save the plunger position
        CRANE_eepromWriteChar(currentMoves[i].plungerTicks, address++);
    }
}

uint8_t doesBufferMatch(volatile connectionBuffer_t buf, uint8_t strictMode, char const * const pStr)
{
    if(buf.readFlag)
    {
        // reset read flag
        buf.readFlag = strictMode ? 0 : buf.readFlag;

        return stringCompare(buf.buffer, pStr);
    }

    return 0;
}

uint8_t stringCompare(char const * const pStrOne, char const * const pStrTwo)
{
    uint8_t i = 0;

    // while string one still has data
    do
    {
        if(*(pStrOne + i) == *(pStrTwo + i))
        {
            // increment
            i++;
        }
        else
        {
            // exit
            return 0;
        }
    } while((*(pStrOne + i) != '\0') && (*(pStrTwo + i) != '\0'));

    // made it out of the loop
    return 1;
}
```

```
/*
 * FileName: StepperMotor.h
 * Version: 1
 *
 * Created: 9/14/2022 2:00 PM
 * Author: Ethan Zeronik
 *
 * Operations: header for the stepper motor submobule
 */

#ifndef StepperMotor_h_INCLUDED
#define StepperMotor_h_INCLUDED

#if defined(__cplusplus)
extern "C" {
#endif

#pragma message("WARNING: this module uses the bottom nibble of the provided port")

#include <stdbool.h>
#include <stdint.h>

/* NOTE: Custom Types */
// typing for the stepper motor enum
typedef enum StepperMotorRunMode_t
{
    // wave step mode
    stepperModeWave = 0,
    // full step mode
    stepperModeFull = 1,
    // half step mode
    stepperModeHalf = 2,
} StepperMotorRunMode_t;

/* NOTE: Function prototypes */
// inits IO for the stepper motor
// takes a pointer to the port to use, assumes botom nibble
void SM_init(uint8_t volatile * const pRegister, uint8_t volatile * const pPort);
// moves the motor in the given mode to the given distance
// distance is in units of rotation
void SM_move(StepperMotorRunMode_t mode, double distance);
// moves the motor in the given mode to the given position
// distance is in units of degrees
void SM_movePosition(StepperMotorRunMode_t mode, uint16_t distance);
// moves the motor in the given mode and the given direction for the given time
// 1 is CW and 0 is CCW
// both times are in ms
void SM_moveTime(StepperMotorRunMode_t mode, bool direction, double time, double stepTime);
// moves the motor in the given mode and the given direction for the given distance
// distance is in steps
// 1 is CW and 0 is CCW
void SM_moveStepsSigned(StepperMotorRunMode_t mode, bool direction, uint16_t distance);

#if defined(__cplusplus)
} /* extern "C" */
#endif
```

```

#endif // StepperMotor_h_INCLUDED
```

```c
/*
 * FileName: StepperMotor.c
 * Version: 1
 *
 * Created: 9/14/2022 2:00 PM
 * Author: Ethan Zeronik
 *
 * Operations: run the stepper motor in one of three modes
 */

/* NOTE: Includes */
#include "StepperMotor.h"

#if !defined(F_CPU)
    #define F_CPU 16000000UL
#endif
// allows for variable delay
#define __DELAY_BACKWARD_COMPATIBLE__

#include <util/delay.h>

/* NOTE: Local declarations */
// local struct for function return
typedef struct StepperMotorModeData_t
{
    // size of the array
    uint8_t               arraySize;
    // pointer to the array
    uint8_t const * const pArray;
    // number of steps to take for desired rotation
    uint32_t              steps;
} StepperMotorModeData_t;
// returns the amount of steps needed for the given mode
// rotation is in radians (I think)
StepperMotorModeData_t getModeAndSteps(StepperMotorRunMode_t mode, double rotation);

/* NOTE: Global Variables */
// implementation of the wave step map
static uint8_t sWaveStepMap[4] = {
    0x01,
    0x02,
    0x04,
    0x08,
};
// implementation of the full step map
static uint8_t sFullStepMap[4] = {
    0x03,
    0x06,
    0x0c,
    0x09,
};
// implementation of the wave step map
static uint8_t sHalfStepMap[8] = {
    0x09,
    0x01,
    0x03,
    0x02,
    0x06,
    0x04,
    0x0c,
    0x08,
};
// instance pointer to the motor port
static uint8_t * sMotorPort;

/* NOTE: Function implementations */
void SM_init(uint8_t volatile * const pRegister, uint8_t volatile * const pPort)
{
    // configure port register
    *pRegister |= 0x0f;

    // turn on pullup resisitors on the bottom nibble
    *pPort = (*pPort & 0xf0) | 0x00;

    // save the port pointer to the static var
    sMotorPort = (uint8_t *)pPort;
}

void SM_move(StepperMotorRunMode_t mode, double distance)
{
    StepperMotorModeData_t data = getModeAndSteps(mode, distance);

    for(uint32_t i = 0, j = 0; i < data.steps; i++)
    {
        *sMotorPort = (*sMotorPort & 0xf0) | data.pArray[j++];

        if(j >= data.arraySize)
        {
            j = 0;
        }

        _delay_ms(3);
    }

    *sMotorPort = *sMotorPort & 0xf0;
}

void SM_movePosition(StepperMotorRunMode_t mode, uint16_t distance)
{
    SM_move(mode, ((double)distance / 360));
}

void SM_moveTime(StepperMotorRunMode_t mode, bool direction, double time, double stepTime)
{
    StepperMotorModeData_t data = getModeAndSteps(mode, 0);

    for(uint32_t i = 0, j = (direction ? data.arraySize : 0); i < (time / stepTime); i++)
    {
        *sMotorPort = (*sMotorPort & 0xf0) | data.pArray[(direction ? j-- : j++)];

        if(j >= data.arraySize || j <= 0)
        {
            j = (direction ? data.arraySize : 0);
        }

        _delay_ms(stepTime);
    }

    *sMotorPort = *sMotorPort & 0xf0;
}

void SM_moveStepsSigned(StepperMotorRunMode_t mode, bool direction, uint16_t distance)
{
    StepperMotorModeData_t data = getModeAndSteps(mode, 0);

    for(uint32_t i = 0, j = (direction ? data.arraySize : 0); i < distance; i++)
    {
        *sMotorPort = (*sMotorPort & 0xf0) | data.pArray[(direction ? j-- : j++)];

        if(j >= data.arraySize || j <= 0)
        {
            j = (direction ? data.arraySize : 0);
        }

        _delay_ms(3);
    }

    *sMotorPort = *sMotorPort & 0xf0;
}

/* NOTE: Local function implementations */
StepperMotorModeData_t getModeAndSteps(StepperMotorRunMode_t mode, double rotation)
{
    uint8_t * pArray;
    uint8_t   size  = 0;
    uint32_t  steps = 0;

    switch(mode)
    {
        case stepperModeWave:
        {
            pArray = sWaveStepMap;
            size   = sizeof(sWaveStepMap) / sizeof(sWaveStepMap[0]);
            steps  = (rotation * 2048);
        }
        break;
        case stepperModeFull:
        {
            pArray = sFullStepMap;
            size   = sizeof(sFullStepMap) / sizeof(sFullStepMap[0]);
            steps  = (rotation * 2048);
        }
        break;
        case stepperModeHalf:
        {
            pArray = sHalfStepMap;
            size   = sizeof(sHalfStepMap) / sizeof(sHalfStepMap[0]);
            steps  = (rotation * 4096);
        }
        break;
        default:
            break;
    };

    return (StepperMotorModeData_t){
        .pArray    = pArray,
        .steps     = steps,
        .arraySize = size,
    };
}
```

```c
/*
 * FileName: AnalogToDigital.h
 * Version: 1
 *
 * Created: 10/19/2022 12:47 AM
 * Author: Ethan Zeronik
 *
 * Operations: header for the adc submodule
 */

#ifndef AnalogToDigital_h_INCLUDED
#define AnalogToDigital_h_INCLUDED

#if defined(__cplusplus)
extern "C" {
#endif

#include <stdint.h>

/* NOTE: Custom Types */
// typing for the handler function
typedef void (*AnalogAsyncGetHandler_t)(uint16_t);

/* NOTE: Function prototypes */
// init registers for adc
void     ADC_init(void);
// init adc for interrupt mode
void     ADC_initInterrupt(void);
// returns the value of the given channel
double   ADC_getTenBitValue(uint16_t channel);
// gets the 10 bit value on the channel
uint16_t ADC_getTenBitValueInterrupt(uint16_t channel);
// set the interrupt handler for the 10 bit async mode
void     ADC_setInterruptHandler(AnalogAsyncGetHandler_t cb);

#if defined(__cplusplus)
} /* extern "C" */
#endif

#endif // AnalogToDigital_h_INCLUDED
```

```c
/*
 * FileName: AnalogToDigital.c
 * Version: 1
 *
 * Created: 10/19/2022 12:47 AM
 * Author: Ethan Zeronik
 *
 * Operations: basic adc implementation
 */

/* NOTE: Includes */
#include "AnalogToDigital.h"

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

/* NOTE: Global Variables */
// value from the interruput
static uint16_t                readInterrupt = 0;
// callback for the interrupt
static AnalogAsyncGetHandler_t interruptCallback;

/* NOTE: Local function implementations */
void ADC_init(void)
{
    // ten bit one way mode
    ADCSRA = (1 << ADEN) | (1 << ADPS1) | (1 << ADPS0);

    // 5v reference
    ADMUX = (1 << REFS0);

    ADCSRB = 0x00;
}

void ADC_initInterrupt(void)
{
    ADC_init();

    ADCSRA |= (1 << ADIE);
}

double ADC_getTenBitValue(uint16_t channel)
{
    uint16_t result = 0;

    // select the channel
    ADMUX  = (ADMUX & 0xe0) | channel;
    ADCSRB = (ADCSRB & 0xf7) | (channel >> 2);

    // start conversion
    ADCSRA |= (1 << ADSC);

    // wait for conversion
    while((ADCSRA & (1 << ADSC)) == 1)
    {
        // do nothing
    }

    // save result
    result = ADCL;
    result = result | (ADCH << 8);

    return result / 1024.0;
}

uint16_t ADC_getTenBitValueInterrupt(uint16_t channel)
{
    // select the channel
    ADMUX  = (ADMUX & 0xe0) | channel;
    ADCSRB = (ADCSRB & 0xf7) | (channel >> 2);

    // start conversion
    ADCSRA |= (1 << ADSC);

    return readInterrupt;
}

void ADC_setInterruptHandler(AnalogAsyncGetHandler_t cb)
{
    interruptCallback = cb;
}

/* NOTE: Local function implementations */
ISR(ADC_vect)
{
    readInterrupt = ADCL;
    readInterrupt = readInterrupt | (ADCH << 8);

    interruptCallback(readInterrupt);
}
```

```c
/*
 * CraneCommunication.h
 *
 * Created: 11/2/2022 8:32:52 AM
 *  Author: xiang82, Alex Weyer, Yu-Hung (Thomas) Wang
 */

#ifndef CraneCommunication_H_INCLUDED
#define CraneCommunication_H_INCLUDED

#if defined(__cplusplus)
extern "C" {
#endif

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

// defines in case of no callback
#define serialInterrupt USART0_RX_vect
#define bluetoothInterrupt USART1_RX_vect

// defines for use in interrupt
#define serialData UDR0
#define bluetoothData UDR1

// sets up usart0 for serial communication
void CRANE_initSerial(uint16_t baudRate);
// sends the given string to the main serialport
void CRANE_sendSerial(char const * const pData);
// sets up usart1 for serial communication
void CRANE_initBluetooth(uint16_t baudRate);
// sends the given string to the bluetooth serialport
void CRANE_sendBluetooth(char const * const pData);

#if defined(__cplusplus)
} /* extern "C" */
#endif

#endif /* CraneCommunication_H_INCLUDED */
```

```c
/*
 * CraneCommunication.c
 *
 * Created: 2022/11/5 10:03:43 am
 * Author: xiang82, Alex Weyer, Yu-Hung (Thomas) Wang
 */

/* NOTE: Includes */
#include "CraneCommunication.h"

#if !defined(F_CPU)
    #define F_CPU 16000000UL
#endif

/* NOTE: Local declarations */
// Xiangs's serial send function
void UART_out(uint8_t ch);
// Alex and Thomas's serial send function
// transmit single byte of data
void BLUETOOTH_out(uint8_t ch);

/* NOTE: Global function implementations */
void CRANE_initSerial(uint16_t baudRate)
{
    // ubrr load
    uint16_t myubr;

    // set up the ucsr0a and ucsr0b and ucsr0c
    UCSR0A = 0x00;
    UCSR0B = (1 << RXCIE0) | (1 << RXEN0) | (1 << TXEN0);
    UCSR0C = (1 << UCSZ00) | (1 << UCSZ01);

    myubr  = (F_CPU / (16UL * (uint16_t)baudRate)) - 1;
    // load ubrr low
    UBRR0L = myubr;

    UBRR0H = 0x00;
}

void CRANE_sendSerial(char const * const pData)
{
    char const * pWorker = (char const *)pData;

    // while we are not at the end of the string
    while(*pWorker != '\0')
    {
        // wait for uart tx to be ready then send out uart
        UART_out(*pWorker);

        pWorker++;
    }
}

void CRANE_initBluetooth(uint16_t baudRate)
{
    uint16_t mybur;
    UCSR1A = 0;

    // enable receive interrupt
    // enable transmits
    // enable receive
    // 2 stop bits
    UCSR1B = (1 << RXCIE1) | (1 << TXEN1) | (1 << RXEN1);

    UCSR1C = (1 << UCSZ11) | (1 << UCSZ10);

    // set up baud rate
    mybur = (F_CPU) / (16UL * (uint16_t)baudRate) - 1;

    UBRR1L = mybur;
    UBRR1H = 0x00;
}

void CRANE_sendBluetooth(char const * const pData)
{
    char const * pWorker = (char const *)pData;

    // while we are not at the end of the string
    while(*pWorker != '\0')
    {
        // wait for uart tx to be ready then send out uart
        BLUETOOTH_out(*pWorker);

        pWorker++;
    }
}

/* NOTE: Local function implementations */
void UART_out(uint8_t ch)
{
    // wait to complete transmission and empty udr0
    while((UCSR0A & (1 << UDRE0)) == 0)
    {
    }

    // load next byte to be transmitted
    UDR0 = ch;
}

void BLUETOOTH_out(uint8_t ch) // transmit single byte of data
{
    while((UCSR1A & (1 << UDRE1)) == 0)
    {
        // wait for completing transmission and empty UDR0
    }

    UDR1 = ch; // load next byte to be transmitted
}
```

```c
/*
 * CraneDelay.h
 *
 * Created: 10/26/2022 8:47:13 AM
 * Author: weyer4
 */

#ifndef CraneDelay_H_INCLUDED
#define CraneDelay_H_INCLUDED

#if defined(__cplusplus)
extern "C" {
#endif

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

// starts timer 0 in async mode
void CRANE_initTimer(void);
// returns the current delay in ms
uint16_t CRANE_tick(void);
// delays for the desired amount of ms
void CRANE_delayMs(uint16_t ms);

#if defined(__cplusplus)
} /* extern "C" */
#endif

#endif /* CraneDelay_H_INCLUDED */
```

```c
/*
 * CraneDelay.c
 *
 * Created: 10/5/2022 8:47:57 AM
 * Author : Alex Weyer
 */

/* NOTE: Includes */
#include "CraneDelay.h"

#if !defined(F_CPU)
    #define F_CPU 16000000UL
#endif

/* NOTE: Local declarations */
// variable to increment and use for delay
volatile uint16_t tick;

/* NOTE: Global function implementations */
void CRANE_initTimer(void)
{
    TCNT0  = 6;
    TCCR0A = 0;
    TCCR0B = 0x03;
    TIMSK0 = (1 << TOIE0);
}

uint16_t CRANE_tick(void)
{
    return tick;
}

void CRANE_delayMs(uint16_t ms){
    uint16_t desiredTick = tick + ms;

    while (desiredTick != tick)
    {
        // do nothing
    }
}

/* NOTE: Local function implementations */
ISR(TIMER0_OVF_vect)
{
    TCNT0 = 6;
    tick++;
}
```

```c
/*
 * CraneEeprom.h
 *
 * Created: 11/2/2022 8:32:52 AM
 *  Author: xiang82
 */

#ifndef CraneEeprom_H_INCLUDED
#define CraneEeprom_H_INCLUDED

#if defined(__cplusplus)
extern "C" {
#endif

#include <avr/io.h>

// write a charater to the given address
void CRANE_eepromWriteChar(char in, uint16_t addr);
// reads a character
char CRANE_eepromReadChar(uint16_t addr);
// write a string to the given address
void CRANE_eepromWriteString(char const * const in, uint16_t addr);
// reads a string from the address into the buffer
void CRANE_eepromReadString(uint16_t addr, char * const buf);

#if defined(__cplusplus)
} /* extern "C" */
#endif

#endif /* CraneEeprom_H_INCLUDED */
```

```c
/*
 * CraneEeprom.c
 *
 * Created: 2022/11/5 10:03:43 am
 * Author: xiang82
 */

/* NOTE: Includes */
#include "CraneEeprom.h"

/* NOTE: Global function implementations */
void CRANE_eepromWriteChar(char ucData, uint16_t uiAddress)
{
    while(EECR & (1 << EEPE))
    {
        /* Wait for completion of previous write */
    }

    /* Set up address and Data Registers */
    EEAR = uiAddress;
    EEDR = ucData;

    /* Write logical one to EEMPE */
    // step 5. write 1 to EEMPE and 0 to EEPE
    EECR = (1 << EEMPE);

    /* Start EEPROM write by setting EEPE */
    // write EEPE within 4 clock cycles
    EECR |= (1 << EEPE);
}

char CRANE_eepromReadChar(uint16_t uiAddress)
{
    while(EECR & (1 << EEPE))
    {
        /* Wait for completion of previous write */
    };

    EEAR = uiAddress;
    EECR |= (1 << EERE);

    return EEDR;
}

void CRANE_eepromWriteString(char const * const ucData, uint16_t uiAddress)
{
    uint16_t n = 0;

    while(ucData[n] != '\0')
    {
        CRANE_eepromWriteChar(ucData[n], uiAddress);

        n++;
    }

    CRANE_eepromWriteChar('\0', uiAddress + 1);
}

void CRANE_eepromReadString(uint16_t uiAddress, char * const EEPROM_buf_ptr)
{
    char * pWorker = (char *)EEPROM_buf_ptr;

    // changed 0xFF to 0x00 because we are looking for end of string
    while(CRANE_eepromReadChar(uiAddress) != 0x00)
    {
        *pWorker = CRANE_eepromReadChar(uiAddress);

        pWorker++;
        uiAddress++;
    }
}
```

```
/*
 * CraneServo.h
 *
 * Created: 2022/11/17 下午 09:01:37
 *  Author: Yu-Hung (Thomas) Wang
 */


#ifndef CraneServo_H_INCLUDED
#define CraneServo_H_INCLUDED

#if defined(__cplusplus)
extern "C" {
#endif

#include <avr/io.h>

// should set up the timer
void CRANE_initServos(void);
// should start the pwm signal back at the previous position
void CRANE_startServos(void);
// should stop the pwm signal and save the position
void CRANE_stopServos(void);
// should change the position of the servo
// if servo = 0 then output compare 0 changes
// if servo = 1 then output comapre 1 changes
// 1ms = 0 and 2ms = 255
void CRANE_setServoPosition(uint8_t servo, uint8_t position);

#if defined(__cplusplus)
} /* extern "C" */
#endif

#endif /* CraneServo_H_INCLUDED */
```

```c
/*
 * CraneServo.c
 *
 * Created: 2022/11/17 下午 09:01:15
 *  Author: Yu-Hung (Thomas) Wang
 */

/* NOTE: Includes */
#include "CraneServo.h"

/* NOTE: Global function implementations */
void CRANE_initServos(void)
{
    // set up PORTB.5 as an output and 0V
    DDRB |= 0x60;
    PORTB |= PORTB & ~0x60;

    // 5000 @ 64
    // set frequency to 50hz
    ICR1 = 5000;

    // fast pwm set on compare
    TCCR1A = 0x02;
    // prescaler set to 64
    TCCR1B = 0x1B;
}

void CRANE_setServoPosition(uint8_t servo, uint8_t position)
{
    if(servo == 0)
    {
        OCR1A = (uint32_t)position * 250 / 255 + 250;
    }
    else if(servo == 1)
    {
        OCR1B = (uint32_t)position * 250 / 255 + 250;
    }
}

void CRANE_startServos(void)
{
    TCCR1A |= 0xA0;
}

void CRANE_stopServos(void)
{
    TCCR1A = (TCCR1A & ~0x80);
}
```