

```
1  /*
2  * FileName: main.c
3  * Version: 1
4  *
5  * Created: 11/11/2022 8:46:12 AM
6  * Author: Ethan Zeronik
7  *
8  * Operations: crane project source code
9  *
10 * Hardware:
11 *   Atmega2560          micro controller
12 *   Port L              LED reposnse bar
13 *   Port C.4            stepper motor limit switch
14 *   Port C.0-3          stepper motor
15 *   Port A.0-3          button inputs
16 *   Port B.5-6          servo outputs
17 */
18
19 /* NOTE: Includes */
20 // standard include for the atmega program
21 #include <avr/io.h>
22 #include <avr/interrupt.h>
23
24 #include <stdio.h>
25
26 // adds stepper motor module (1/6)
27 #include "StepperMotor.h"
28 // adds adc for input potentiometers (2/6)
29 #include "AnalogToDigital.h"
30 // adds pwm for servo motors (3/6)
31 #include "CraneServo.h"
32 // adds UART and Bluetooth for communication (4/6)
33 #include "CraneCommunication.h"
34 // adds EEPROM read/write for persistant data storage (5/6)
35 #include "CraneEeprom.h"
36 // adds custom delay functions (6/6)
37 #include "CraneDelay.h"
38
39 /* NOTE: Types and Structs */
40 // type to hold data for serial connections
41 typedef struct connectionBuffer_t
42 {
43     // the buffer to store the data in
44     char    buffer[128];
45     // the current index of the last set byte in the buffer
46     uint8_t index;
47     // whether or not the buffer is done being written to
48     uint8_t readFlag;
49 } connectionBuffer_t;
50 // type to hold the position of the servos and motor
51 typedef struct cranePosition_t
52 {
53     // motor position in steps
54     int16_t motorTicks;
55     // arm position in 0-255 ticks
56     uint8_t armTicks;
```

```

57     // plunger position in 0-255 ticks
58     uint8_t plungerTicks;
59 } cranePosition_t;
60
61 /* NOTE: Custom Macros */
62 // the three main states of the application
63 #define calibrateState 0
64 #define actionState 1
65 #define recordState 2
66
67 // command definition
68 #define recordModeCommand "Calibrate"
69 #define calibrateModeCommand "Reset"
70 #define runCommand "Run"
71 #define recordCommand "Record"
72 #define getCommand "Get"
73
74 // custom buttons & switches
75 #define limitSwitch (PINC & 0x10)
76 #define leftButton (PINA & 0x01)
77 #define rightButton (PINA & 0x02)
78 #define recordButton (PINA & 0x04)
79 #define calibrateButton (PINA & 0x08)
80
81 // custom led bar
82 #define stateLed PORTL
83
84 // custom servos ids
85 #define armServo 0
86 #define plungerServo 1
87
88 // defined values for where default spots of the servo are
89 #define armStartPosition 150
90 #define plungerStartPosition 230
91
92 // record information
93 #define recordLength 6
94 #define eepromAddress 0x0555
95
96 /* NOTE: Global Variables */
97 // the state of the application
98 uint8_t applicationState = calibrateState;
99 // the state of all the moving parts of the crane
100 cranePosition_t craneState = {
101     .motorTicks = 0,
102     .armTicks = 0,
103     .plungerTicks = 0,
104 };
105 // the EEPROM recorded moves to make
106 cranePosition_t recordedMoves[recordLength];
107 // the moves to save to eeprom
108 cranePosition_t currentMoves[recordLength];
109 // current spot in the instance moves
110 uint8_t currentMoveIndex = 0;
111 // the input buffer for serial
112 volatile connectionBuffer_t serialInputData = {

```

```
113     .buffer    = {0},
114     .index     = 0,
115     .readFlag  = 0,
116 };
117 // the input buffer for bluetooth
118 volatile connectionBuffer_t bluetoothInputData = {
119     .buffer    = {0},
120     .index     = 0,
121     .readFlag  = 0,
122 };
123
124 /* NOTE: Function prototypes */
125 // inits IO ports
126 void IO_init(void);
127 // gets the saved values from the eeprom
128 void CRANE_getMovesFromEeprom(uint16_t addr);
129 // saves the current recorded moves to eeprom
130 void CRANE_saveMovesToEeprom(uint16_t addr);
131 // takes a string and checks if the buffer matches the value exactly
132 // strictmode resets readflag
133 // returns 1 if true, else 0
134 uint8_t doesBufferMatch(volatile connectionBuffer_t buf, uint8_t strictMode, char const *
const pStr);
135 // compares to strings
136 // if the match it returns 1, else 0
137 uint8_t stringCompare(char const * const pStrOne, char const * const pStrTwo);
138
139 /* NOTE: Application implementation */
140 // the main loop of the function, provided to us
141 int main(void)
142 {
143     // start the gpio
144     IO_init();
145
146     // init the ADC and the servo control
147     ADC_init();
148     CRANE_initServos();
149     CRANE_startServos();
150
151     // in this case, 1 is CCW, and 0 is CW
152     // pass in the port and register we want to use for the motor
153     SM_init(&DDRC, &PORTC);
154
155     // get the stored recorded data from the EEPROM
156     CRANE_getMovesFromEeprom(eepromAddress);
157
158     // turn on the serial on port 0 at 9600 baud
159     CRANE_initSerial(9600);
160     CRANE_sendSerial("Crane online\r\n");
161
162     // turn on the bluetooth connection
163     CRANE_initBluetooth(9600);
164     CRANE_sendBluetooth("Crane online\r\n");
165
166     // start the delay timer
167     CRANE_initTimer();
```

```
168
169 // start global interrupts
170 sei();
171
172 while(1)
173 {
174     if(blueetoothInputData.readFlag)
175     {
176         CRANE_sendSerial(blueetoothInputData.buffer);
177     }
178
179     // DEBUG: get the state
180     // if we get the command for record mode and we are not already in record mode
181     if((doesBufferMatch(serialInputData, 0, recordModeCommand) && (applicationState !=
recordState)) || (doesBufferMatch(blueetoothInputData, 0, recordModeCommand) &&
(applicationState != recordState)) || (recordButton && (applicationState != recordState)))
182     {
183         CRANE_sendSerial("Entering record mode...\r\n");
184         CRANE_sendBluetooth("Entering record mode...\r\n");
185
186         applicationState = recordState;
187         serialInputData.readFlag = 0;
188         blueetoothInputData.readFlag = 0;
189
190         while(recordButton)
191         {
192             // do nothing until we let go
193         }
194     }
195     // check for reset command
196     else if((doesBufferMatch(serialInputData, 0, calibrateModeCommand) &&
(applicationState != calibrateState)) || (doesBufferMatch(blueetoothInputData, 1,
calibrateModeCommand) && (applicationState != calibrateState)) || (calibrateButton &&
(applicationState != calibrateState)))
197     {
198         CRANE_sendSerial("Resetting...\r\n");
199         CRANE_sendBluetooth("Resetting...\r\n");
200
201         applicationState = calibrateState;
202         serialInputData.readFlag = 0;
203         blueetoothInputData.readFlag = 0;
204
205         while(calibrateButton)
206         {
207             // do nothing until we let go
208         }
209     }
210
211     // display state on leds
212     stateLed = (stateLed & 0xfc) | applicationState;
213
214     // DEBUG: check for the get command
215     if(doesBufferMatch(serialInputData, 0, getCommand) ||
doesBufferMatch(blueetoothInputData, 0, getCommand))
216     {
217         char response[96];
218         sprintf(response, "Motor is at %i, Plunger is at %u, Arm is at %u\r\n",
craneState.motorTicks, craneState.plungerTicks, craneState.armTicks);
```

```

219
220     serialInputData.readFlag    = 0;
221     bluetoothInputData.readFlag = 0;
222
223     CRANE_sendSerial(response);
224     CRANE_sendBluetooth(response);
225 }
226
227 // main application switch case
228 switch(applicationState)
229 {
230     // DEBUG: the action case
231     case actionState:
232     {
233         if(leftButton || doesBufferMatch(serialInputData, 0, runCommand) ||
doesBufferMatch(bluetoothInputData, 0, runCommand))
234         {
235             // HACK: potential multiple select
236             serialInputData.readFlag    = 0;
237             bluetoothInputData.readFlag = 0;
238
239             for(uint8_t i = 0; i < recordLength; i++)
240             {
241                 char response[32];
242                 sprintf(response, "Running recorded step %u...\r\n", i + 1);
243
244                 CRANE_sendSerial(response);
245                 CRANE_sendBluetooth(response);
246                 stateLed = (stateLed & 0x03) | 1 << (i + 2);
247
248                 // calculate the relative movement of the arm
249                 int16_t moveSteps = recordedMoves[i].motorTicks -
craneState.motorTicks;
250
251                 // move motor
252                 if(moveSteps > 0)
253                 {
254                     SM_moveStepsSigned(stepperModeHalf, 0, moveSteps);
255                 }
256                 else if(moveSteps < 0)
257                 {
258                     SM_moveStepsSigned(stepperModeHalf, 1, -1 * moveSteps);
259                 }
260
261                 // lerp!
262                 for(uint8_t j = 1; j < 101; j++)
263                 {
264                     CRANE_setServoPosition(armServo, craneState.armTicks +
((recordedMoves[i].armTicks - craneState.armTicks) * ((float)j / 100)));
265                     CRANE_delayMs(5);
266                 }
267
268                 for(uint8_t j = 1; j < 101; j++)
269                 {
270                     CRANE_setServoPosition(plungerServo, craneState.plungerTicks +
((recordedMoves[i].plungerTicks - craneState.plungerTicks) * ((float)j / 100)));
271                     CRANE_delayMs(5);

```

```
272     }
273
274     // set our state
275     craneState.motorTicks = recordedMoves[i].motorTicks;
276     craneState.armTicks   = recordedMoves[i].armTicks;
277     craneState.plungerTicks = recordedMoves[i].plungerTicks;
278
279     CRANE_delayMs(100);
280 }
281
282 // reset state led
283 stateLed = (stateLed & 0x03);
284
285 while(leftButton)
286 {
287     // do nothing until we let go
288 }
289 }
290 }
291 break;
292
293 // DEBUG: the record case
294 case recordState:
295 {
296     uint8_t armPosition      = 255 * ADC_getTenBitValue(0);
297     uint8_t plungerPosition = 255 * ADC_getTenBitValue(1);
298     int16_t moveSteps        = 8;
299
300     // display the step we are recording
301     stateLed = (stateLed & 0x03) | 1 << (currentMoveIndex + 2);
302
303     // manually move the motor
304     if(rightButton)
305     {
306         SM_moveStepsSigned(stepperModeHalf, 0, moveSteps);
307         CRANE_delayMs(10);
308     }
309     else if(leftButton)
310     {
311         SM_moveStepsSigned(stepperModeHalf, 1, moveSteps);
312         CRANE_delayMs(10);
313
314         // negate for current position
315         moveSteps *= -1;
316     }
317     else
318     {
319         moveSteps = 0;
320     }
321
322     // manually move the servos
323     CRANE_setServoPosition(armServo, armPosition);
324     CRANE_setServoPosition(plungerServo, plungerPosition);
325
326     // update the current positions
327     craneState.motorTicks += moveSteps;
```

```

328         craneState.armTicks      = armPosition;
329         craneState.plungerTicks = plungerPosition;
330
331         // if we press the record button save the position
332         if(recordButton || doesBufferMatch(serialInputData, 0, recordCommand) ||
doesBufferMatch(bluetoothInputData, 0, recordCommand))
333         {
334             serialInputData.readFlag = 0;
335             bluetoothInputData.readFlag = 0;
336
337             char response[64];
338             sprintf(response, "Recording step %u out of 6...\r\n", currentMoveIndex +
1);
339
340             CRANE_sendSerial(response);
341             CRANE_sendBluetooth(response);
342
343             sprintf(response, "Recorded {%i,%u,%u}...\r\n", craneState.motorTicks,
craneState.plungerTicks, craneState.armTicks);
344
345             CRANE_sendSerial(response);
346             CRANE_sendBluetooth(response);
347
348             // reset state led
349             stateLed = (stateLed & 0x03);
350
351             if(currentMoveIndex < recordLength)
352             {
353                 currentMoves[currentMoveIndex++] = craneState;
354             }
355
356             if(currentMoveIndex >= recordLength)
357             {
358                 CRANE_saveMovesToEeprom(eepromAddress);
359                 currentMoveIndex = 0;
360
361                 // done recording, back to action state after zeroing
362                 // home ---> action(play)
363                 applicationState = calibrateState;
364             }
365
366             while(recordButton)
367             {
368                 // do nothing until we let go
369             }
370         }
371     }
372     break;
373
374     // DEBUG: the default case will be the home case
375     case calibrateState:
376     default:
377     {
378         // 0 is up 255 is down
379         CRANE_setServoPosition(armServo, armStartPosition);
380         // 255 is close 0 is open
381         CRANE_setServoPosition(plungerServo, plungerStartPosition);

```

```
382
383 // move CW for one second to ensure the limit switch is not set
384 SM_moveTime(stepperModeHalf, 0, 1000, 3);
385
386 // while not hitting the switch
387 while(!limitSwitch)
388 {
389     // then move CCW a bit at the time until we hit the limit switch
390     SM_moveStepsSigned(stepperModeHalf, 1, 24);
391 }
392
393 // then move 30 degrees back to center the arm
394 SM_movePosition(stepperModeHalf, 35);
395
396 // set the current position
397 craneState.motorTicks = 0;
398 craneState.armTicks = armStartPosition;
399 craneState.plungerTicks = plungerStartPosition;
400
401 // then set it to action state
402 applicationState = actionState;
403 }
404 break;
405 }
406 }
407 }
408
409 // interrupt handling for the serial connection
410 ISR(serialInterrupt)
411 {
412     if(serialData != '\r' && serialData != '\n' && serialData != '\0' &&
413 (serialInputData.index < 127))
414     {
415         // add to array
416         serialInputData.buffer[serialInputData.index] = serialData;
417         serialInputData.buffer[serialInputData.index + 1] = '\0';
418         serialInputData.index++;
419     }
420     else
421     {
422         // set update flag
423         serialInputData.readFlag = 1;
424         // reset message index
425         serialInputData.index = 0;
426     }
427 }
428
429 // interrupt handling for the bluetooth connection
430 ISR(blueetoothInterrupt)
431 {
432     if(blueetoothData != '\r' && blueetoothData != '\n' && blueetoothData != '\0' &&
433 (blueetoothInputData.index < 127))
434     {
435         // add to array
436         blueetoothInputData.buffer[blueetoothInputData.index] = blueetoothData;
437         blueetoothInputData.buffer[blueetoothInputData.index + 1] = '\0';
```



```
437     bluetoothInputData.index++;
438 }
439 else
440 {
441     // set update flag
442     bluetoothInputData.readFlag = 1;
443     // reset message index
444     bluetoothInputData.index = 0;
445 }
446 }
447 }
448
449 /* NOTE: Function implementations */
450 void IO_init(void)
451 {
452     // port c.4 is the limit switch
453     DDRC = 0x00;
454     PORTC = 0x10;
455
456     // port a.0-1 are for left and right
457     DDRA = 0x00;
458     PORTA = 0xff;
459
460     DDRL = 0xff;
461     PORTL = 0x00;
462 }
463
464 void CRANE_getMovesFromEeprom(uint16_t addr)
465 {
466     uint16_t address = addr;
467
468     for(uint8_t i = 0; i < recordLength; i++)
469     {
470         uint16_t motorTicks = 0;
471
472         // read the motor position
473         motorTicks = CRANE_eepromReadChar(address++) << 8;
474         motorTicks += CRANE_eepromReadChar(address++);
475
476         recordedMoves[i].motorTicks = motorTicks;
477
478         // read the arm position
479         recordedMoves[i].armTicks = CRANE_eepromReadChar(address++);
480
481         // read the plunger position
482         recordedMoves[i].plungerTicks = CRANE_eepromReadChar(address++);
483     }
484 }
485
486 void CRANE_saveMovesToEeprom(uint16_t addr)
487 {
488     uint16_t address = addr;
489
490     for(uint8_t i = 0; i < recordLength; i++)
491     {
492         // cheat by directly moving into our recorded array
```

```
493     recordedMoves[i] = currentMoves[i];
494
495     // save the motor position
496     CRANE_eepromWriteChar((currentMoves[i].motorTicks & 0xff00) >> 8, address++);
497     CRANE_eepromWriteChar((currentMoves[i].motorTicks & 0x00ff), address++);
498
499     // save the arm position
500     CRANE_eepromWriteChar(currentMoves[i].armTicks, address++);
501
502     // save the plunger position
503     CRANE_eepromWriteChar(currentMoves[i].plungerTicks, address++);
504 }
505 }
506
507 uint8_t doesBufferMatch(volatile connectionBuffer_t buf, uint8_t strictMode, char const *
const pStr)
508 {
509     if(buf.readFlag)
510     {
511         // reset read flag
512         buf.readFlag = strictMode ? 0 : buf.readFlag;
513
514         return stringCompare(buf.buffer, pStr);
515     }
516
517     return 0;
518 }
519
520 uint8_t stringCompare(char const * const pStrOne, char const * const pStrTwo)
521 {
522     uint8_t i = 0;
523
524     // while string one still has data
525     do
526     {
527         if(*(pStrOne + i) == *(pStrTwo + i))
528         {
529             // increment
530             i++;
531         }
532         else
533         {
534             // exit
535             return 0;
536         }
537     } while((*(pStrOne + i) != '\0') && (*(pStrTwo + i) != '\0'));
538
539     // made it out of the loop
540     return 1;
541 }
```

```

1  /*
2   * FileName: StepperMotor.h
3   * Version: 1
4   *
5   * Created: 9/14/2022 2:00 PM
6   * Author: Ethan Zeronik
7   *
8   * Operations: header for the stepper motor submobule
9   */
10
11 #ifndef StepperMotor_h_INCLUDED
12 #define StepperMotor_h_INCLUDED
13
14 #if defined(__cplusplus)
15 extern "C" {
16 #endif
17
18 #pragma message("WARNING: this module uses the bottom nibble of the provided port")
19
20 #include <stdbool.h>
21 #include <stdint.h>
22
23 /* NOTE: Custom Types */
24 // typing for the stepper motor enum
25 typedef enum StepperMotorRunMode_t
26 {
27     // wave step mode
28     stepperModeWave = 0,
29     // full step mode
30     stepperModeFull = 1,
31     // half step mode
32     stepperModeHalf = 2,
33 } StepperMotorRunMode_t;
34
35 /* NOTE: Function prototypes */
36 // inits IO for the stepper motor
37 // takes a pointer to the port to use, assumes botom nibble
38 void SM_init(uint8_t volatile * const pRegister, uint8_t volatile * const pPort);
39 // moves the motor in the given mode to the given distance
40 // distance is in units of rotation
41 void SM_move(StepperMotorRunMode_t mode, double distance);
42 // moves the motor in the given mode to the given position
43 // distance is in units of degrees
44 void SM_movePosition(StepperMotorRunMode_t mode, uint16_t distance);
45 // moves the motor in the given mode and the given direction for the given time
46 // 1 is CW and 0 is CCW
47 // both times are in ms
48 void SM_moveTime(StepperMotorRunMode_t mode, bool direction, double time, double stepTime);
49 // moves the motor in the given mode and the given direction for the given distance
50 // distance is in steps
51 // 1 is CW and 0 is CCW
52 void SM_moveStepsSigned(StepperMotorRunMode_t mode, bool direction, uint16_t distance);
53
54 #if defined(__cplusplus)
55 } /* extern "C" */
56 #endif

```

```
57 |  
58 | #endif // StepperMotor_h_INCLUDED
```

```

1  /*
2  * FileName: StepperMotor.c
3  * Version: 1
4  *
5  * Created: 9/14/2022 2:00 PM
6  * Author: Ethan Zeronik
7  *
8  * Operations: run the stepper motor in one of three modes
9  */
10
11 /* NOTE: Includes */
12 #include "StepperMotor.h"
13
14 #if !defined(F_CPU)
15     #define F_CPU 16000000UL
16 #endif
17 // allows for variable delay
18 #define __DELAY_BACKWARD_COMPATIBLE__
19
20 #include <util/delay.h>
21
22 /* NOTE: Local declarations */
23 // local struct for function return
24 typedef struct StepperMotorModeData_t
25 {
26     // size of the array
27     uint8_t      arraySize;
28     // pointer to the array
29     uint8_t const * const pArray;
30     // number of steps to take for desired rotation
31     uint32_t      steps;
32 } StepperMotorModeData_t;
33 // returns the amount of steps needed for the given mode
34 // rotation is in radians (I think)
35 StepperMotorModeData_t getModeAndSteps(StepperMotorRunMode_t mode, double rotation);
36
37 /* NOTE: Global Variables */
38 // implementation of the wave step map
39 static uint8_t sWaveStepMap[4] = {
40     0x01,
41     0x02,
42     0x04,
43     0x08,
44 };
45 // implementation of the full step map
46 static uint8_t sFullStepMap[4] = {
47     0x03,
48     0x06,
49     0x0c,
50     0x09,
51 };
52 // implementation of the wave step map
53 static uint8_t sHalfStepMap[8] = {
54     0x09,
55     0x01,
56     0x03,

```

```
57     0x02,  
58     0x06,  
59     0x04,  
60     0x0c,  
61     0x08,  
62 };  
63 // instance pointer to the motor port  
64 static uint8_t * sMotorPort;  
65  
66 /* NOTE: Function implementations */  
67 void SM_init(uint8_t volatile * const pRegister, uint8_t volatile * const pPort)  
68 {  
69     // configure port register  
70     *pRegister |= 0x0f;  
71  
72     // turn on pullup resisitors on the bottom nibble  
73     *pPort = (*pPort & 0xf0) | 0x00;  
74  
75     // save the port pointer to the static var  
76     sMotorPort = (uint8_t *)pPort;  
77 }  
78  
79 void SM_move(StepperMotorRunMode_t mode, double distance)  
80 {  
81     StepperMotorModeData_t data = getModeAndSteps(mode, distance);  
82  
83     for(uint32_t i = 0, j = 0; i < data.steps; i++)  
84     {  
85         *sMotorPort = (*sMotorPort & 0xf0) | data.pArray[j++];  
86  
87         if(j >= data.arraySize)  
88         {  
89             j = 0;  
90         }  
91  
92         _delay_ms(3);  
93     }  
94  
95     *sMotorPort = *sMotorPort & 0xf0;  
96 }  
97  
98 void SM_movePosition(StepperMotorRunMode_t mode, uint16_t distance)  
99 {  
100     SM_move(mode, ((double)distance / 360));  
101 }  
102  
103 void SM_moveTime(StepperMotorRunMode_t mode, bool direction, double time, double stepTime)  
104 {  
105     StepperMotorModeData_t data = getModeAndSteps(mode, 0);  
106  
107     for(uint32_t i = 0, j = (direction ? data.arraySize : 0); i < (time / stepTime); i++)  
108     {  
109         *sMotorPort = (*sMotorPort & 0xf0) | data.pArray[(direction ? j-- : j++)];  
110  
111         if(j >= data.arraySize || j <= 0)  
112         {
```

```
113         j = (direction ? data.arraySize : 0);
114     }
115
116     _delay_ms(stepTime);
117 }
118
119 *sMotorPort = *sMotorPort & 0xf0;
120 }
121
122 void SM_moveStepsSigned(StepperMotorRunMode_t mode, bool direction, uint16_t distance)
123 {
124     StepperMotorModeData_t data = getModeAndSteps(mode, 0);
125
126     for(uint32_t i = 0, j = (direction ? data.arraySize : 0); i < distance; i++)
127     {
128         *sMotorPort = (*sMotorPort & 0xf0) | data.pArray[(direction ? j-- : j++)];
129
130         if(j >= data.arraySize || j <= 0)
131         {
132             j = (direction ? data.arraySize : 0);
133         }
134
135         _delay_ms(3);
136     }
137
138     *sMotorPort = *sMotorPort & 0xf0;
139 }
140
141 /* NOTE: Local function implementations */
142 StepperMotorModeData_t getModeAndSteps(StepperMotorRunMode_t mode, double rotation)
143 {
144     uint8_t * pArray;
145     uint8_t size = 0;
146     uint32_t steps = 0;
147
148     switch(mode)
149     {
150     case stepperModeWave:
151     {
152         pArray = sWaveStepMap;
153         size = sizeof(sWaveStepMap) / sizeof(sWaveStepMap[0]);
154         steps = (rotation * 2048);
155     }
156     break;
157     case stepperModeFull:
158     {
159         pArray = sFullStepMap;
160         size = sizeof(sFullStepMap) / sizeof(sFullStepMap[0]);
161         steps = (rotation * 2048);
162     }
163     break;
164     case stepperModeHalf:
165     {
166         pArray = sHalfStepMap;
167         size = sizeof(sHalfStepMap) / sizeof(sHalfStepMap[0]);
168         steps = (rotation * 4096);
```

```
169     }
170     break;
171     default:
172         break;
173 };
174
175 return (StepperMotorModeData_t){
176     .pArray    = pArray,
177     .steps     = steps,
178     .arraySize = size,
179 };
180 }
```



```
1  /*
2   * FileName: AnalogToDigital.h
3   * Version: 1
4   *
5   * Created: 10/19/2022 12:47 AM
6   * Author: Ethan Zeronik
7   *
8   * Operations: header for the adc submodule
9   */
10
11 #ifndef AnalogToDigital_h_INCLUDED
12 #define AnalogToDigital_h_INCLUDED
13
14 #if defined(__cplusplus)
15 extern "C" {
16 #endif
17
18 #include <stdint.h>
19
20 /* NOTE: Custom Types */
21 // typing for the handler function
22 typedef void (*AnalogAsyncGetHandler_t)(uint16_t);
23
24 /* NOTE: Function prototypes */
25 // init registers for adc
26 void ADC_init(void);
27 // init adc for interrupt mode
28 void ADC_initInterrupt(void);
29 // returns the value of the given channel
30 double ADC_getTenBitValue(uint16_t channel);
31 // gets the 10 bit value on the channel
32 uint16_t ADC_getTenBitValueInterrupt(uint16_t channel);
33 // set the interrupt handler for the 10 bit async mode
34 void ADC_setInterruptHandler(AnalogAsyncGetHandler_t cb);
35
36 #if defined(__cplusplus)
37 } /* extern "C" */
38 #endif
39
40 #endif // AnalogToDigital_h_INCLUDED
```

```
1  /*
2   * FileName: AnalogToDigital.c
3   * Version: 1
4   *
5   * Created: 10/19/2022 12:47 AM
6   * Author: Ethan Zeronik
7   *
8   * Operations: basic adc implementation
9   */
10
11 /* NOTE: Includes */
12 #include "AnalogToDigital.h"
13
14 #include <avr/io.h>
15 #include <avr/interrupt.h>
16
17 /* NOTE: Global Variables */
18 // value from the interrupt
19 static uint16_t readInterrupt = 0;
20 // callback for the interrupt
21 static AnalogAsyncGetHandler_t interruptCallback;
22
23 /* NOTE: Local function implementations */
24 void ADC_init(void)
25 {
26     // ten bit one way mode
27     ADCSRA = (1 << ADEN) | (1 << ADPS1) | (1 << ADPS0);
28
29     // 5v reference
30     ADMUX = (1 << REFS0);
31
32     ADCSRB = 0x00;
33 }
34
35 void ADC_initInterrupt(void)
36 {
37     ADC_init();
38
39     ADCSRA |= (1 << ADIE);
40 }
41
42 double ADC_getTenBitValue(uint16_t channel)
43 {
44     uint16_t result = 0;
45
46     // select the channel
47     ADMUX = (ADMUX & 0xe0) | channel;
48     ADCSRB = (ADCSRB & 0xf7) | (channel >> 2);
49
50     // start conversion
51     ADCSRA |= (1 << ADSC);
52
53     // wait for conversion
54     while((ADCSRA & (1 << ADSC)) == 1)
55     {
56         // do nothing
```

```
57     }
58
59     // save result
60     result = ADCL;
61     result = result | (ADCH << 8);
62
63     return result / 1024.0;
64 }
65
66 uint16_t ADC_getTenBitValueInterrupt(uint16_t channel)
67 {
68     // select the channel
69     ADMUX = (ADMUX & 0xe0) | channel;
70     ADCSRB = (ADCSRB & 0xf7) | (channel >> 2);
71
72     // start conversion
73     ADCSRA |= (1 << ADSC);
74
75     return readInterrupt;
76 }
77
78 void ADC_setInterruptHandler(AnalogAsyncGetHandler_t cb)
79 {
80     interruptCallback = cb;
81 }
82
83 /* NOTE: Local function implementations */
84 ISR(ADC_vect)
85 {
86     readInterrupt = ADCL;
87     readInterrupt = readInterrupt | (ADCH << 8);
88
89     interruptCallback(readInterrupt);
90 }
```

```
1  /*
2   * CraneCommunication.h
3   *
4   * Created: 11/2/2022 8:32:52 AM
5   * Author: xiang82, Alex Weyer, Yu-Hung (Thomas) Wang
6   */
7
8  #ifndef CraneCommunication_H_INCLUDED
9  #define CraneCommunication_H_INCLUDED
10
11  #if defined(__cplusplus)
12  extern "C" {
13  #endif
14
15  #include <avr/io.h>
16  #include <avr/interrupt.h>
17
18  // defines in case of no callback
19  #define serialInterrupt USART0_RX_vect
20  #define bluetoothInterrupt USART1_RX_vect
21
22  // defines for use in interrupt
23  #define serialData UDR0
24  #define bluetoothData UDR1
25
26  // sets up usart0 for serial communication
27  void CRANE_initSerial(uint16_t baudRate);
28  // sends the given string to the main serialport
29  void CRANE_sendSerial(char const * const pData);
30  // sets up usart1 for serial communication
31  void CRANE_initBluetooth(uint16_t baudRate);
32  // sends the given string to the bluetooth serialport
33  void CRANE_sendBluetooth(char const * const pData);
34
35  #if defined(__cplusplus)
36  } /* extern "C" */
37  #endif
38
39  #endif /* CraneCommunication_H_INCLUDED */
```

```
1  /*
2  * CraneCommunication.c
3  *
4  * Created: 2022/11/5 10:03:43 am
5  * Author: xiang82, Alex Weyer, Yu-Hung (Thomas) Wang
6  */
7
8  /* NOTE: Includes */
9  #include "CraneCommunication.h"
10
11  #if !defined(F_CPU)
12      #define F_CPU 16000000UL
13  #endif
14
15  /* NOTE: Local declarations */
16  // Xiangs's serial send function
17  void UART_out(uint8_t ch);
18  // Alex and Thomas's serial send function
19  // transmit single byte of data
20  void BLUETOOTH_out(uint8_t ch);
21
22  /* NOTE: Global function implementations */
23  void CRANE_initSerial(uint16_t baudRate)
24  {
25      // ubrr load
26      uint16_t myubr;
27
28      // set up the ucsr0a and ucsr0b and ucsr0c
29      UCSR0A = 0x00;
30      UCSR0B = (1 << RXCIE0) | (1 << RXEN0) | (1 << TXEN0);
31      UCSR0C = (1 << UCSZ00) | (1 << UCSZ01);
32
33      myubr = (F_CPU / (16UL * (uint16_t)baudRate)) - 1;
34      // load ubrr low
35      UBRR0L = myubr;
36
37      UBRR0H = 0x00;
38  }
39
40  void CRANE_sendSerial(char const * const pData)
41  {
42      char const * pWorker = (char const *)pData;
43
44      // while we are not at the end of the string
45      while(*pWorker != '\0')
46      {
47          // wait for uart tx to be ready then send out uart
48          UART_out(*pWorker);
49
50          pWorker++;
51      }
52  }
53
54  void CRANE_initBluetooth(uint16_t baudRate)
55  {
56      uint16_t mybur;
```

```
57     UCSR1A = 0;
58
59     // enable receive interrupt
60     // enable transmits
61     // enable receive
62     // 2 stop bits
63     UCSR1B = (1 << RXCIE1) | (1 << TXEN1) | (1 << RXEN1);
64
65     UCSR1C = (1 << UCSZ11) | (1 << UCSZ10);
66
67     // set up baud rate
68     mybur = (F_CPU) / (16UL * (uint16_t)baudRate) - 1;
69
70     UBRR1L = mybur;
71     UBRR1H = 0x00;
72 }
73
74 void CRANE_sendBluetooth(char const * const pData)
75 {
76     char const * pWorker = (char const *)pData;
77
78     // while we are not at the end of the string
79     while(*pWorker != '\0')
80     {
81         // wait for uart tx to be ready then send out uart
82         BLUETOOTH_out(*pWorker);
83
84         pWorker++;
85     }
86 }
87
88 /* NOTE: Local function implementations */
89 void UART_out(uint8_t ch)
90 {
91     // wait to complete transmission and empty udr0
92     while((UCSR0A & (1 << UDRE0)) == 0)
93     {
94     }
95
96     // load next byte to be transmitted
97     UDR0 = ch;
98 }
99
100 void BLUETOOTH_out(uint8_t ch) // transmit single byte of data
101 {
102     while((UCSR1A & (1 << UDRE1)) == 0)
103     {
104         // wait for completing transmission and empty UDR0
105     }
106
107     UDR1 = ch; // load next byte to be transmitted
108 }
```

```
1  /*
2   * CraneDelay.h
3   *
4   * Created: 10/26/2022 8:47:13 AM
5   * Author: weyer4
6   */
7
8  #ifndef CraneDelay_H_INCLUDED
9  #define CraneDelay_H_INCLUDED
10
11  #if defined(__cplusplus)
12  extern "C" {
13  #endif
14
15  #include <avr/interrupt.h>
16  #include <avr/io.h>
17
18  // starts timer 0 in async mode
19  void CRANE_initTimer(void);
20  // returns the current delay in ms
21  uint16_t CRANE_tick(void);
22  // delays for the desired amount of ms
23  void CRANE_delayMs(uint16_t ms);
24
25  #if defined(__cplusplus)
26  } /* extern "C" */
27  #endif
28
29  #endif /* CraneDelay_H_INCLUDED */
```

```
1  /*
2   * CraneDelay.c
3   *
4   * Created: 10/5/2022 8:47:57 AM
5   * Author : Alex Weyer
6   */
7
8  /* NOTE: Includes */
9  #include "CraneDelay.h"
10
11 #if !defined(F_CPU)
12     #define F_CPU 16000000UL
13 #endif
14
15 /* NOTE: Local declarations */
16 // variable to increment and use for delay
17 volatile uint16_t tick;
18
19 /* NOTE: Global function implementations */
20 void CRANE_initTimer(void)
21 {
22     TCNT0 = 6;
23     TCCR0A = 0;
24     TCCR0B = 0x03;
25     TIMSK0 = (1 << TOIE0);
26 }
27
28 uint16_t CRANE_tick(void)
29 {
30     return tick;
31 }
32
33 void CRANE_delayMs(uint16_t ms){
34     uint16_t desiredTick = tick + ms;
35
36     while (desiredTick != tick)
37     {
38         // do nothing
39     }
40 }
41
42 /* NOTE: Local function implementations */
43 ISR(TIMER0_OVF_vect)
44 {
45     TCNT0 = 6;
46     tick++;
47 }
```



```
1  /*
2   * CraneEeprom.h
3   *
4   * Created: 11/2/2022 8:32:52 AM
5   * Author: xiang82
6   */
7
8  #ifndef CraneEeprom_H_INCLUDED
9  #define CraneEeprom_H_INCLUDED
10
11  #if defined(__cplusplus)
12  extern "C" {
13  #endif
14
15  #include <avr/io.h>
16
17  // write a character to the given address
18  void CRANE_eepromWriteChar(char in, uint16_t addr);
19  // reads a character
20  char CRANE_eepromReadChar(uint16_t addr);
21  // write a string to the given address
22  void CRANE_eepromWriteString(char const * const in, uint16_t addr);
23  // reads a string from the address into the buffer
24  void CRANE_eepromReadString(uint16_t addr, char * const buf);
25
26  #if defined(__cplusplus)
27  } /* extern "C" */
28  #endif
29
30  #endif /* CraneEeprom_H_INCLUDED */
```

```
1  /*
2  * CraneEeprom.c
3  *
4  * Created: 2022/11/5 10:03:43 am
5  * Author: xiang82
6  */
7
8  /* NOTE: Includes */
9  #include "CraneEeprom.h"
10
11 /* NOTE: Global function implementations */
12 void CRANE_eepromWriteChar(char ucData, uint16_t uiAddress)
13 {
14     while(EECR & (1 << EEPE))
15     {
16         /* Wait for completion of previous write */
17     }
18
19     /* Set up address and Data Registers */
20     EEAR = uiAddress;
21     EEDR = ucData;
22
23     /* Write logical one to EEMPE */
24     // step 5. write 1 to EEMPE and 0 to EEPE
25     EECR = (1 << EEMPE);
26
27     /* Start EEPROM write by setting EEPE */
28     // write EEPE within 4 clock cycles
29     EECR |= (1 << EEPE);
30 }
31
32 char CRANE_eepromReadChar(uint16_t uiAddress)
33 {
34     while(EECR & (1 << EEPE))
35     {
36         /* Wait for completion of previous write */
37     };
38
39     EEAR = uiAddress;
40     EECR |= (1 << EERE);
41
42     return EEDR;
43 }
44
45 void CRANE_eepromWriteString(char const * const ucData, uint16_t uiAddress)
46 {
47     uint16_t n = 0;
48
49     while(ucData[n] != '\0')
50     {
51         CRANE_eepromWriteChar(ucData[n], uiAddress);
52
53         n++;
54     }
55
56     CRANE_eepromWriteChar('\0', uiAddress + 1);
```

```
57 | }
58 |
59 | void CRANE_eepromReadString(uint16_t uiAddress, char * const EEPROM_buf_ptr)
60 | {
61 |     char * pWorker = (char *)EEPROM_buf_ptr;
62 |
63 |     // changed 0xFF to 0x00 because we are looking for end of string
64 |     while(CRANE_eepromReadChar(uiAddress) != 0x00)
65 |     {
66 |         *pWorker = CRANE_eepromReadChar(uiAddress);
67 |
68 |         pWorker++;
69 |         uiAddress++;
70 |     }
71 | }
```

```
1  /*
2   * CraneServo.h
3   *
4   * Created: 2022/11/17 下午 09:01:37
5   * Author: Yu-Hung (Thomas) Wang
6   */
7
8
9  #ifndef CraneServo_H_INCLUDED
10 #define CraneServo_H_INCLUDED
11
12 #if defined(__cplusplus)
13 extern "C" {
14 #endif
15
16 #include <avr/io.h>
17
18 // should set up the timer
19 void CRANE_initServos(void);
20 // should start the pwm signal back at the previous position
21 void CRANE_startServos(void);
22 // should stop the pwm signal and save the position
23 void CRANE_stopServos(void);
24 // should change the position of the servo
25 // if servo = 0 then output compare 0 changes
26 // if servo = 1 then output compare 1 changes
27 // 1ms = 0 and 2ms = 255
28 void CRANE_setServoPosition(uint8_t servo, uint8_t position);
29
30 #if defined(__cplusplus)
31 } /* extern "C" */
32 #endif
33
34 #endif /* CraneServo_H_INCLUDED */
```

```
1  /*
2   * CraneServo.c
3   *
4   * Created: 2022/11/17 下午 09:01:15
5   * Author: Yu-Hung (Thomas) Wang
6   */
7
8  /* NOTE: Includes */
9  #include "CraneServo.h"
10
11 /* NOTE: Global function implementations */
12 void CRANE_initServos(void)
13 {
14     // set up PORTB.5 as an output and 0V
15     DDRB |= 0x60;
16     PORTB |= PORTB & ~0x60;
17
18     // 5000 @ 64
19     // set frequency to 50hz
20     ICR1 = 5000;
21
22     // fast pwm set on compare
23     TCCR1A = 0x02;
24     // prescaler set to 64
25     TCCR1B = 0x1B;
26 }
27
28 void CRANE_setServoPosition(uint8_t servo, uint8_t position)
29 {
30     if(servo == 0)
31     {
32         OCR1A = (uint32_t)position * 250 / 255 + 250;
33     }
34     else if(servo == 1)
35     {
36         OCR1B = (uint32_t)position * 250 / 255 + 250;
37     }
38 }
39
40 void CRANE_startServos(void)
41 {
42     TCCR1A |= 0xA0;
43 }
44
45 void CRANE_stopServos(void)
46 {
47     TCCR1A = (TCCR1A & ~0x80);
48 }
```