```
 1: // required device includes/settings
 2: #include <18F46K22.h>
 3: #device adc=10
 4: #device high_ints=TRUE
 5:
 6: // standard libraries
 7: #include <stdio.h>
 8: #include <stdint.h>
 9: #include <stdlib.h>
10: #include <string.h>
11: #include <stdlibm.h>
12:
13: // confinguration files
14: #include "pic_config.h"
15: #include "defines.h"
16: #include "function_headers.h"
17: #include "jack_dn2500.h"
18: #include "globals.h"
19:
20: // specific headers
21: #include "pic.h"
22: #include "dust.h"
23: #include "periph.h"
24: #include "control.h"
25: #include "valve.h"
26: #include "battery.h"
27: #include "stacks_queues.h"
28: #include "util.h"
29:
30: void main()
31: {
32:    uint8_t     priority_queue_item_to_exectue = EMPTY_PRIORITY_QUEUE;
33:
34:    // set system state to init
35:    global_system_state = SYSTEM_INIT;
36:    // initalize oscillator and timing of rs232, i2c, delay
37:    osc_init();
38:    // initalize all variables
39:    //vars_init();
40:
41:    // grab all eeprom values (e.g. vlv cal, sprinkler number, vlv position)
42:    read_all_eeprom_values();
43:
44:    // initalize all periphs, timers, ccps
45:    periph_init();
46:
```

```
47:    // reinitalizes all of the global variables to their defaults
48:    //vars_init();
49:
50:    // clear queues
51:    clear_priority_queue();
52:    clear_time_queue();
53:    clear_message_queue();
54:    //allocate_command_queue();
55:    //clear_command_queue();
56:
57:    // startup rtc, turn on gen_rpm, enable dust
58:    start_rtc();
59:    setup_gen_rpm();
60:    mote_init();
61:
62:    // if cold start was done, reset mote
63:    if (global_previous_shutdown_cause == COLD_RESTART_REQUEST)
64:    {
65:       mote_reset();
66:    }
67:
68:    // DELETE?
69:    enable_interrupts(GLOBAL);
70:
71:    // if SW1 is asserted during boot, trigger searching alogrithm
72:    if (!input(SW1n))
73:    {
74:       // set the system state up for searching for a network (not run)
75:       global_system_state = SYSTEM_SEARCHING_FOR_NETWORK;
76:       strcpy (global_temp_line_buff, "Setup Manager...");
77:       LCD_line1(global_temp_line_buff);
78:       strcpy (global_temp_line_buff, "  Release SW1   ");
79:       LCD_line2(global_temp_line_buff);
80:       // wait for switch release
81:       while (!input(SW1n));
82:       // delay lcd update for a tiny bit
83:       global_skip_lcd_update_count = 1;
84:       // first part of search alorithm sequence
85:       PUSH_PRIORITY_QUEUE_MACRO(SEARCH_FOR_STRONGEST_1);
86:    }
87:
88:    // normal behavior, check mote status.  Join network if necessary.
89:    else
90:    {
91:       // Fun startup splash screen
92:       //LCD_startup_splash();
```

```
 93:       //LCD_clear();
 94:       // dispaly battery voltage and restart/shutdown cause on screen for a second
 95:       LCD_display_battery_voltage(0);
 96:       LCD_display_shutdown_cause(1);
 97:       global_skip_lcd_update_count = 2;
 98:       // all periphs should be initalized (except mote), so system should be ready to run     fdafdsafdsafdsafdsaf
 99:       global_system_state = SYSTEM_RUN;
100:       // check mote state. If it needs to connect, it will react as it should
101:       PUSH_TIME_QUEUE_MACRO(global_rtc_time + 5, CHECK_MOTE_STATE);
102:
103:    }
104:
105:    // start the queue/control loop timer
106:    setup_T2_int(T2_64MS);
107:    //setup_T0_int(T0_1S);
108:    //set_mppc(global_mppc_value);
109:    //global_charge_duty = (MAX_CHARGE - NO_CHARGE)/2;
110:    //global_charge_duty = MAX_CHARGE;
111:    //set_charge_duty(global_charge_duty);
112:
113:    //PUSH_PRIORITY_QUEUE_MACRO(CALIBRATE_VALVE_1);
114:    while(1)
115:    {
116: ////////////////////Start of Priority Queue Handling//////////////////////////
117:
118:       // Timer 2 is used to signal the priority queue to check for another item
119:       //    to execute. This is only done once every 64ms to not keep the
120:       //    system in a state with the interrupts disabled all the time.
121:       // Timer 2 is also used to time the control loop.  How often the set point
122:       //    is re-evaluated and adjustments are made to the actual values is
123:       //    controlled by how many control_loop_delay_cycles_left
124:       if (TMR2IF)
125:       {
126:          // reset the timer 2 queue loop interrupt
127:          setup_T2_int(T2_64MS);
128:
129:          update_control_loop();
130:          // Safely grab the next item in the priority queue to execute
131:          // All interrupts that can modify the queue need to be disabled to
132:          //    ensure memory is not corrupted.
133:          disable_interrupts(INT_CCP4);
134:          priority_queue_item_to_exectue = pop_priority_queue();
135:          enable_interrupts(INT_CCP4);
136:
137:          // execute the priority queue item
138:          switch (priority_queue_item_to_exectue)
```

```
139:          {
140:             // Decode a new packet and react/respond appropriately
141:             case DEAL_WITH_NEW_PACKET:
142:                // Display that you're dealing with a new packet
143:                LCD_clear();
144:                strcpy (global_temp_line_buff, "Deal With Packet");
145:                LCD_line1(global_temp_line_buff);
146:                global_skip_lcd_update_count = 2;
147:
148:                // disable ccp4 and mote interrupt so we don't overwrite payload_buff
149:                //    or have colliding unsolicited messages
150:                disable_interrupts(INT_CCP4);
151:                disable_interrupts(INT_EXT2_H2L);
152:                deal_with_packet();
153:                enable_interrupts(INT_EXT2_H2L);
154:                enable_interrupts(INT_CCP4);
155:                break;
156:
157:             // Start the somewhat convoluted calibrate valve routine
158:             // 1. Open valve VLV_CAL_1_MOVEMENT w/ "starting current"
159:             // 2. Close valve fully w/ current being "normal closing current"
160:             // 3. Open valve fully w/ normal current regimes
161:             // 4. Close valve fully w/ normal current regimes
162:             // 5. Send valve calibration response to mote
163:             // steps and system states are handled in COMP and CCP3 ISR as well
164:             //    as setting the calibration values
165:
166:             case CALIBRATE_VALVE_1:
167:                // Change system state: initial open for calibrate valve routine
168:                global_system_state = SYSTEM_CAL_VLV_1;
169:                // display calibration routine on screen
170:                LCD_clear();
171:                strcpy (global_temp_line_buff, "VLV Calibration ");
172:                LCD_line1(global_temp_line_buff);
173:                strcpy (global_temp_line_buff, "Begining........");
174:                LCD_line2(global_temp_line_buff);
175:                global_skip_lcd_update_count = 2;
176:                // setup brakes and charging for valve calibration
177:                // (maximum resistance with no RPM Control)
178:                global_control_loop_mechanism = NO_RPM_CONTROL_DYN_MPPC;
179:                global_charge_duty_set_value = MAX_CHARGE;
180:                global_brake_duty_set_value = MAX_BRK;
181:                // put values to default values (45 seconds open/close)
182:                global_valve_time_to_close_1024th = DEFAULT_VLV_TIME_TO_CLOSE;
183:                global_valve_time_to_open_1024th = DEFAULT_VLV_TIME_TO_OPEN;
184:                // Set the global valve position to the default value (middle)
```

```
185:                // this gives the valve a reference point to open a little from
186:                global_valve_position = VLV_PRECALIBRAION_POSITION;
187:                // Set valve position slightly more open than it is and move valve
188:                global_valve_position_set_value = (VLV_PRECALIBRAION_POSITION +   \
189:                VLV_CAL_1_MOVEMENT);
190:                PUSH_PRIORITY_QUEUE_MACRO(MOVE_VALVE);
191:                break;
192:
193:             case CALIBRATE_VALVE_2:
194:                // Change system state: fully closed
195:                global_system_state = SYSTEM_CAL_VLV_2;
196:                // set valve position target to fully closed and move valve
197:                global_valve_position_set_value = VLV_POSITION_CLOSED;
198:                PUSH_PRIORITY_QUEUE_MACRO(MOVE_VALVE);
199:                break;
200:
201:             case CALIBRATE_VALVE_3:
202:                // Change system state: fully closed to fully opened
203:                global_system_state = SYSTEM_CAL_VLV_3;
204:                // set valve position target to fully open and move valve
205:                global_valve_position_set_value = VLV_POSITION_OPENED;
206:                PUSH_PRIORITY_QUEUE_MACRO(MOVE_VALVE);
207:                break;
208:
209:            case CALIBRATE_VALVE_4:
210:                // Change system state: FSR (not used at the moment)
211:                global_system_state = SYSTEM_CAL_VLV_4;
212:                // move to the next calibration stage
213:                PUSH_PRIORITY_QUEUE_MACRO(CALIBRATE_VALVE_5);
214:                break;
215:
216:             case CALIBRATE_VALVE_5:
217:                // Change system state: fully opened to fully closed
218:                global_system_state = SYSTEM_CAL_VLV_5;
219:                // set valve position target to fully closed and move valve
220:                global_valve_position_set_value = VLV_POSITION_CLOSED;
221:                PUSH_PRIORITY_QUEUE_MACRO(MOVE_VALVE);
222:                break;
223:
224:             case CALIBRATE_VALVE_6:
225:                // Change system state: display calibration and send to manager
226:                global_system_state = SYSTEM_CAL_VLV_6;
227:                // display calibraion on screen
228:                LCD_clear();
229:                strcpy (global_temp_line_buff, "Close CCP=      ");
230:                LCD_line1(global_temp_line_buff);
```

```
231:                strcpy (global_temp_line_buff, "Open CCP =      ");
232:                LCD_line2(global_temp_line_buff);
233:                LCD_place_uint16(global_valve_time_to_close_1024th,0,11,5);
234:                LCD_place_uint16(global_valve_time_to_open_1024th,1,11,5);
235:                // put calibration stuff on screen for 4 seconds
236:                global_skip_lcd_update_count = 5;
237:                // if valve calibration time is below the limit, it triggers an error
238:                //    and throws away the calibration, returning it to the run state.
239:                if ((global_valve_time_to_close_1024th < ERROR_VLV_CAL_TIME) || \
240:                   (global_valve_time_to_open_1024th < ERROR_VLV_CAL_TIME))
241:                {
242:                   // reset valve calibration times to the defaults
243:                   global_valve_time_to_open_1024th = DEFAULT_VLV_TIME_TO_OPEN;
244:                   global_valve_time_to_close_1024th = DEFAULT_VLV_TIME_TO_CLOSE;
245:                   // change valve position to unknown
246:                   global_valve_position = VLV_POSITION_UNKNOWN;
247:                   // set the error bitfield and send an error
248:                   global_error_message_bitfield |= ERR_MSG_VLV_CAL_FAIL;
249:                   PUSH_MESSAGE_QUEUE_MACRO(MSG_MOTE_ERROR_MSG);
250:                }
251:                // successful/valid calibration time
252:                else
253:                {
254:                   // update the calibration time
255:                   global_valve_calibration_utc_time = global_utc_time;
256:                   // store calibrations in eeprom
257:                   store_vcal_eeprom_values();
258:                   // send an unsolicited valve report to the manager
259:                   PUSH_MESSAGE_QUEUE_MACRO(MSG_MOTE_VALVE_REPORT);
260:                }
261:                // put system in run state
262:                global_system_state = SYSTEM_RUN;
263:                break;
264:
265:             // move valve to position specified by calibrate FSR routine
266:             case CALIBRATE_FSR_1:
267:                // set system state
268:                global_system_state = SYSTEM_CAL_FSR_1;
269:                // save the current valve, so we can return to it later
270:
271:                // move valve to the FSR position
272:                global_valve_position_set_value = global_calibrate_fsr_valve_position;
273:                PUSH_PRIORITY_QUEUE_MACRO(MOVE_VALVE);
274:                break;
275:
276:             case CALIBRATE_FSR_2:
```

```
277:                // set system state
278:                global_system_state = SYSTEM_CAL_FSR_2;
279:                // actually measure the FSR and store it
280:                global_calibrate_fsr_period = global_current_period;
281:                PUSH_MESSAGE_QUEUE_MACRO(MSG_MOTE_VALVE_REPORT);
282:
283:                global_system_state = SYSTEM_RUN;
284:                break;
285:
286:             case CALIBRATE_FSR_3:
287:                global_system_state = SYSTEM_CAL_FSR_3;
288:                break;
289:             /*
290:             case CALIBRATE_FSR_4:
291:                global_system_state = SYSTEM_CAL_FSR_4;
292:                LCD_clear();
293:                LCD_place_uint16(global_valve_position,0,0,5);
294:                LCD_place_uint16(global_valve_time_to_close_1024th,0,6,5);
295:                LCD_place_uint16(global_valve_time_to_open_1024th,0,11,5);
296:                LCD_place_uint16(global_valve_position,1,0,5);
297:                LCD_place_uint16(global_valve_position,1,11,5);
298:                LCD_place_uint16(global_valve_position,1,11,5);
299:                // put calibration stuff on screen for 4 seconds
300:                global_skip_lcd_update_count = 5;
301:                global_calibrate_fsr_utc_time
302:
303:                global_system_state = SYSTEM_RUN;
304:                break;
305:                */
306:             case MOVE_VALVE_MAG_DECOUPLING_RECOVERY:
307:                // move the valve to VLV_MAGNETIC_COUPLING_FIX
308:                global_valve_position_set_value = VLV_MAGNETIC_COUPLING_FIX;
309:                PUSH_PRIORITY_QUEUE_MACRO(MOVE_VALVE);
310:                // change control loop scheme
311:                global_control_loop_mechanism = MAG_DECOUPLING_RECOVERY;
312:                break;
313:
314:             case MOVE_VALVE_NO_SPIN_RECOVERY:
315:                // move the valve to VLV_NOT_SPIN_FIX
316:                global_valve_position_set_value = VLV_NOT_SPIN_FIX;
317:                PUSH_PRIORITY_QUEUE_MACRO(MOVE_VALVE);
318:                // change control loop scheme
319:                global_control_loop_mechanism = NO_SPIN_RECOVERY;
320:                break;
321:
322:             // Moves valve to global_valve_position_set_value (set before calling
```

```
323:            //    this.)
324:            case MOVE_VALVE:
325:               // switch to 1Mhz clock as the interrupt overhead is too high for
326:               //    reliable valve movement @ 250k
327:               //fosc_1m();
328:
329:               // if system is in an undesireable state, don't move the valve
330:               //    and send an error message
331:               // Undesirable states such as unkown or init
332:               if ((global_system_state == SYSTEM_STATE_UNKNOWN)||            \
333:                  (global_system_state == SYSTEM_INIT))
334:               {
335:                  global_error_message_bitfield |= ERR_MSG_INCOMPATIBLE_STATE;
336:                  PUSH_MESSAGE_QUEUE_MACRO(MSG_MOTE_ERROR_MSG);
337:                  break;
338:               }
339:               // or if you're in the run state and the valve is uncalibrated
340:               else if ((global_system_state == SYSTEM_RUN) &&                \
341:                  (global_valve_position == VLV_POSITION_UNKNOWN))
342:               {
343:                  global_error_message_bitfield |= ERR_MSG_VLV_NOT_CALIBRATED;
344:                  PUSH_MESSAGE_QUEUE_MACRO(MSG_MOTE_ERROR_MSG);
345:                  break;
346:               }
347:
348:               // if valve movement is not needed (less than one millispan away
349:               //    and in run state), break out of switch case
350:               if ((global_system_state == SYSTEM_RUN) &&                     \
351:               (((global_valve_position_set_value - global_valve_position) < 0x20) || \
352:               ((global_valve_position - global_valve_position_set_value) < 0x20)))
353:               {
354:                  break;
355:               }
356:
357:               // clear out the time in motion
358:               global_valve_time_in_motion_1024ths = 0;
359:
360:               // increment the movements since hitting an endstop. Also, check if
361:               // enough valve movements have happened to warrant a recalibration of
362:               //    the position of the valve by going towards an endstop
363:               if ((global_valve_movements_since_endstop++) > VLV_MOVES_BEFORE_RECAL)
364:               {
365:                  global_system_state = SYSTEM_RECAL_VLV_MOVES;
366:                  // quicker to go to the closed endstop. Start motion.
367:                  if ((global_valve_position_set_value + global_valve_position) >\
368:                     VLV_POSITION_OPENED)
```

```
369:                  {
370:                     CCP_3 = global_valve_time_to_close_1024th;
371:                     mV_CLOSEm;
372:                  }
373:                  // quicker to go to the open endstop. Start motion.
374:                  else
375:                  {
376:                     CCP_3 = global_valve_time_to_open_1024th;
377:                     // don't go to the open endstop to avoid magnetic decoupling
378:                     //mV_OPENm;
379:                     mV_CLOSEm;
380:                  }
381:               }
382:
383:               // regular valve move (without recalibration)
384:               else
385:               {
386:                  // setup CCP3 and put the valve into motion based on target value
387:                  if (global_valve_position_set_value > global_valve_position)
388:                  {
389:                     CCP_3 = global_valve_time_to_open_1024th;
390:                     mV_OPENm;
391:                  }
392:                  else if (global_valve_position_set_value < global_valve_position)
393:                  {
394:                     CCP_3 = global_valve_time_to_close_1024th;
395:                     mV_CLOSEm;
396:                  }
397:                  else if (global_valve_position_set_value == VLV_POSITION_OPENED)
398:                  {
399:                     CCP_3 = global_valve_time_to_open_1024th;
400:                     mV_OPENm;
401:                  }
402:                  else if (global_valve_position_set_value == VLV_POSITION_CLOSED)
403:                  {
404:                     CCP_3 = global_valve_time_to_close_1024th;
405:                     mV_CLOSEm;
406:                  }
407:               }
408:
409:               // setup/turn on the comparator interrupt (also sets up DAC)
410:               comparator_setup();
411:
412:               // setup and enable CCP3 as well as it's respective timer
413:               setup_ccp3(CCP_USE_TIMER1_AND_TIMER2 | CCP_COMPARE_RESET_TIMER);
414:               setup_timer_1(T1_ENABLE_SOSC | T1_EXTERNAL_SYNC | T1_DIV_BY_1);
```

```
415:               set_timer1(0);
416:
417:               // clear any ccp3 interrupt and enable ccp3 interrupt
418:               clear_interrupt(INT_CCP3);
419:               enable_interrupts(INT_CCP3);
420:               break;
421:
422:            // Start GPS aquisition
423:            case START_GPS_AQUISITION:
424:               break;
425:
426:            // Check GPS for lock
427:            case CHECK_GPS_FOR_LOCK:
428:               break;
429:
430:            // Update the LCD (with the time for now)
431:            case LCD_UPDATE:
432:               // instructed to skip this update (to show other things on screen)
433:               if (global_skip_lcd_update_count > 1)
434:               {
435:                  global_skip_lcd_update_count--;
436:               }
437:               // either normal update or screen clear and update
438:               else
439:               {
440:                  // last update was skipped, clear screen and fill screen
441:                  if (global_skip_lcd_update_count == 1)
442:                  {
443:                     LCD_clear();
444:                     global_skip_lcd_update_count = 0;
445:                     strcpy (global_temp_line_buff, "V      S    B   ");
446:                     LCD_line1(global_temp_line_buff);
447:                     strcpy (global_temp_line_buff, "T      M    C   ");
448:                     LCD_line2(global_temp_line_buff);
449:                  }
450:
451:                  LCD_place_uint16(global_valve_position, 0, 1, 5);
452:                  LCD_place_uint16(global_current_rpm,0,8,3);
453:                  LCD_place_uint32(global_rtc_time, 1,1,5);
454:                  LCD_place_uint8(global_current_message_queue_location, 1, 7, 3);
455:                  //LCD_place_uint32(global_utc_time, 1,1,10);
456:                  /*
457:                  LCD_place_uint8(global_current_message_queue_location, 1, 8, 3);
458:                  if (global_current_message_queue_location != 255)
459:                  {
460:                     LCD_place_uint8(global_message_queue[global_current_message_queue_location].message_type,0,13
```

```
461:                     LCD_place_uint8(global_message_queue[global_current_message_queue_location].attempt_num,1,13,
462:                  }
463:                  */
464:                  LCD_place_uint16(global_brake_duty,0,13,3);
465:                  LCD_place_uint16(global_charge_duty,1,13,3);
466:                  /*
467:                  get_vgen(0);
468:                  LCD_place_uint16(global_brake_duty, 0, 0, 5);
469:                  LCD_place_uint16(global_vgen_set_value, 0, 6, 4);
470:                  LCD_place_uint16(global_vgen, 0, 12, 4);
471:                  LCD_place_uint16(max_batt_charge_current, 1, 0, 4);
472:                  LCD_place_uint8(max_charge_mppc, 1, 5, 3);
473:                  LCD_place_uint8(global_mppc_value, 1, 9, 3);
474:                  LCD_place_uint16(calc_gen_rpm(), 1, 13, 3);
475:                  */
476:                  /*
477:                  LCD_place_uint8(global_mppc_value, 1, 0, 3);
478:                  LCD_place_uint8(global_rpm_set_value, 1, 5, 3);
479:                  LCD_place_uint16(calc_gen_rpm(), 1, 11, 5);
480:                  */
481:               }
482:               break;
483:
484:            // recovers from an i2c bus collision interrupt
485:            case BUSCOL_RESET:
486:               BCL1IF = FALSE;
487:               SSP1IF = TRUE;
488:               i2c_init(TRUE);
489:               if (global_lcd_enabled) LCD_init();
490:               enable_interrupts(GLOBAL);
491:               break;
492:
493:            // Clear the LCD (usually on a time queue)
494:            case LCD_CLEAR_SCREEN:
495:               lcd_clear();
496:               break;
497:
498:            // Reset the mote (triggers a wait for boot event)
499:            case RESET_MOTE:
500:               strcpy (global_temp_line_buff, "Resetting Mote!!");
501:               LCD_line1(global_temp_line_buff);
502:               global_skip_lcd_update_count = 2;
503:               global_dust_enabled = 0;
504:               mote_reset();
505:               break;
506:
```

```
507:            // Make sure the mote is responding to a boot or shutdown
508:            case WAIT_FOR_BOOT_EVENT:
509:               if (global_dust_enabled == 0)
510:               {
511:                  LCD_clear();
512:                  strcpy (global_temp_line_buff, "Mote is Dead    ");
513:                  LCD_line1(global_temp_line_buff);
514:                  // save shutdown cause and queue shutdown
515:                  global_shutdown_cause = ERR_FAIL_ON_MOTE_RESET;
516:                  PUSH_PRIORITY_QUEUE(SHUTDOWN_SYSTEM);
517:               }
518:               break;
519:
520:            // Check the mote status and react appropriately
521:            case CHECK_MOTE_STATE:
522:               mote_state_check();
523:               break;
524:
525:            case UPDATE_MOTE_TIME:
526:               mote_time_update();
527:               break;
528:
529:            case UPDATE_MOTE_NETWORK_INFO:
530:               get_mote_mac_address();
531:               break;
532:
533:            // Check's battery voltage, decides to charge, not charge, tell manager
534:            // about a low voltage state, or to go to deep sleep
535:            case CHECK_BATTERY_STATE:
536:               /*
537:               LCD_clear();
538:               LCD_display_battery_voltage(0);
539:               LCD_place_uint16(get_vbatt(0),1,0,5);
540:               global_skip_lcd_update_count = 2;
541:               */
542:               check_and_deal_with_battery();
543:               break;
544:
545:            // Query the mote for the temp and store it
546:            case CHECK_MOTE_TEMP:
547:               mote_temp_check();
548:               /*
549:               LCD_clear();
550:               strcpy (global_temp_line_buff, "Temp =         C");
551:               LCD_line1(global_temp_line_buff);
552:               LCD_place_uint8(global_mote_temperature,0,7,3);
```

```
553:                global_skip_lcd_update_count = 2;
554:                */
555:                break;
556:
557:             // First part of the search for strongest algorithm
558:             case SEARCH_FOR_STRONGEST_1:
559:                LCD_clear();
560:                strcpy (global_temp_line_buff, "Search Strong 1 ");
561:                LCD_line1(global_temp_line_buff);
562:                global_skip_lcd_update_count = 2;
563:                // set the state of the system appropriately
564:                global_system_state = SYSTEM_SEARCHING_FOR_NETWORK;
565:                PUSH_PRIORITY_QUEUE_MACRO(RESET_MOTE);
566:                PUSH_TIME_QUEUE_MACRO((global_rtc_time + 10), SEARCH_FOR_STRONGEST_2);
567:                break;
568:
569:             // Second part of the search for strongest algorithm
570:             case SEARCH_FOR_STRONGEST_2:
571:                LCD_clear();
572:                strcpy (global_temp_line_buff, "Search Strong 2 ");
573:                LCD_line1(global_temp_line_buff);
574:                global_skip_lcd_update_count = 2;
575:                search_for_strongest();
576:                break;
577:
578:             // Initalizes a mote join
579:             case INIT_JOIN:
580:                LCD_clear();
581:                strcpy (global_temp_line_buff, "  Init Join     ");
582:                LCD_line1(global_temp_line_buff);
583:                global_skip_lcd_update_count = 2;
584:                initiate_join();
585:                break;
586:
587:             case OPTIMIZE_MPPC:
588:                adjust_mppc();
589:                break;
590:
591:             // resets the cpu (if all pending messages are sent)
592:             case CPU_RESET:
593:                // if the dust network is operational and the message queue is not empty (location at 255)
594:                //    wait for the message to be ack'd/resent and reschedule the shutdown.
595:                if ((global_dust_operational == TRUE) && (global_current_message_queue_location != 255))
596:                {
597:                   PUSH_TIME_QUEUE_MACRO((global_rtc_time + 10), CPU_RESET);
598:                }
```

```
599:                // if the valve is moving, check again later
600:                else if (!IS_VLV_COASTING)
601:                {
602:                   PUSH_TIME_QUEUE_MACRO((global_rtc_time + 10), CPU_RESET);
603:                }
604:                // All messages are sent
605:                else
606:                {
607:                   // if it does, restart the entire system
608:                   store_all_eeprom_values();
609:                   reset_cpu();
610:                }
611:                break;
612:
613:             // Shutsdown the pic for a variety of reasons
614:             case SHUTDOWN_SYSTEM:
615:                // if the dust network is operational and the message queue is not empty (location at 255)
616:                //    wait for the message to be ack'd/resent and reschedule the shutdown.
617:                // I suppose this has potential to be problamatic, but the network should eventually show
618:                //    up as non-operational in mote-check or get ack'd at some point, I would hope.
619:
620:                global_brake_duty_set_value = NO_BRK;
621:                global_charge_duty_set_value = NO_CHARGE;
622:                global_control_loop_mechanism = NO_RPM_CONTROL_DYN_MPPC;
623:
624:                if ((global_brake_duty != NO_BRK) || (global_charge_duty != NO_CHARGE))
625:                {
626:                   break;
627:                }
628:                else if ((global_dust_operational == TRUE) && (global_current_message_queue_location != 255))
629:                {
630:                   PUSH_TIME_QUEUE_MACRO((global_rtc_time + 10), SHUTDOWN_SYSTEM);
631:                }
632:                // if the valve is moving, check again later
633:                else if (!IS_VLV_COASTING)
634:                {
635:                   PUSH_TIME_QUEUE_MACRO((global_rtc_time + 10), SHUTDOWN_SYSTEM);
636:                }
637:                // All messages are sent or system is shutting down due to no network connection
638:                else if ((global_system_state == SYSTEM_RUN) || ((global_system_state == SYSTEM_RUN) && (global_dus
639:                {
640:                   // try to put the mote to sleep
641:                   if (mote_sleep() == NO_ERR)
642:                   {
643:                      // if it does, put the entire system to sleep
644:                      store_all_eeprom_values();
```

```
645:                     deep_sleep();
646:                  }
647:                  // otherwise, try again in 10 seconds
648:                  else
649:                  {
650:                     PUSH_TIME_QUEUE_MACRO((global_rtc_time + 10), SHUTDOWN_SYSTEM);
651:                  }
652:               }
653:               // Anything else, just wait
654:               else
655:               {
656:                  PUSH_TIME_QUEUE_MACRO((global_rtc_time + 10), SHUTDOWN_SYSTEM);
657:               }
658:               break;
659:
660:            // updates
661:            // Default case (nothing to do)
662:            case EMPTY_PRIORITY_QUEUE:
663:               //LCD_place_uint32(global_rtc_time, 1, 0, 10);
664:               break;
665:         }
666:
667:       }
668: ////////////////////End of Priority Queue Handling//////////////////////////
669:    // end of inifinite while loop
670:    }
671: }
672:
673:
674: //#PRIORITY COMP, CCP5, EXT2, CCP3, CCP4, BUSCOL
675: #PRIORITY CCP5, COMP, CCP3, EXT2, TIMER0, CCP4, BUSCOL, RB, EXT
676:
677: /*
678: #INT_RB
679: void RB_ISR(void)
680: {
681:    // only RB6 can be causing this interrupt
682:    // SW1n has changed state
683:
684:    static uint32_t sw1_start_time = 0, sw1_end_time = 0;
685:
686:    if (!input(SW1n))
687:    {
688:       sw1_start_time = global_rtc_time;
689:    }
690:    else
```

```
691:    {
692:       if ((global_rtc_time - sw1_start_time) > SW1_ISR_LONG_PRESS_TIME)
693:       {
694:          //reset ??
695:       }
696:       else
697:       {
698:          disable_interrupts(GLOBAL);
699:          push_priority_queue_ISR(LCD_UPDATE);
700:          enable_interrupts(GLOBAL);
701:          //update LCD with next screen??
702:       }
703:    }
704: }
705: */
706:
707: #INT_EXT
708: void vgen_wakeup_ISR(void)
709: {
710: // runs on wakeup from vgen int
711:    reset_cpu();
712: }
713: #INT_BUSCOL
714: void BUSCOL_ISR(void)
715: {
716: // Catches and recovers from an i2c bus collision
717:
718:    // are reenabled in MAIN
719:    disable_interrupts(GLOBAL);
720:
721:    // clear bus collision interrupt flag
722:    BCL1IF = FALSE;
723:
724:    // schedule a bus collision reset
725:    PUSH_PRIORITY_QUEUE_ISR_MACRO(BUSCOL_RESET);
726:
727:    // if the lcd is connected, reset it
728:    if (global_lcd_enabled)
729:    {
730:       output_low(LCD_RESETn);
731:       delay_cycles(64);       // about 1 millisecond
732:       output_high(LCD_RESETn);
733:    }
734:
735:    // send the stack pointer to position 1 (perhaps sort of dangerous)
736:    STKPTR = 1;
```

```
737: }
738:
739: #INT_EXT2
740: void mote_interrupt(void)
741: {
742: // interrupt called when mote rts line gets asserted
743:    uint8_t     tmp_oscccon, tmp_t2con, tmp_pr2, tmp_t0con;
744:
745:    // save the current oscillator setup
746:    tmp_oscccon = OSCCON;
747:    tmp_t2con = T2CON;
748:    tmp_pr2 = PR2;
749:    tmp_t0con = T0CON;
750:
751:    // start primary (3.8Mhz) crystal for UART communication
752:    fosc_pri_ISR();
753:
754:    // recieve serial data, respond, and schedule deal with packet if necessary
755:    deal_with_mote_ISR();
756:
757:    // restore the current oscillator
758:    OSCCON = tmp_oscccon;
759:    T2CON = tmp_t2con;
760:    T0CON = tmp_t0con;
761:    PR2 = tmp_PR2;
762: }
763:
764:
765: #INT_COMP
766: void comp1_ISR(void)
767: {
768: // ISR routine that is called when the comparator current limit is reached
769:    uint16_t    temp16_frac;
770:
771:     // if comarator 1 is tripped (INT_COMP is triggered by comp 1 or 2)
772:    if (C1OUT)
773:    {
774:       // grab the extra time/2ndary osc ticks since the last 1024th interrupt
775:       temp16_frac = get_timer1();
776:
777:       // if statements for different calibration routines
778:       // valve opening a little bit to ensure we dont jam into close endstop
779:       if (global_system_state ==  SYSTEM_CAL_VLV_1)
780:       {
781:          global_valve_position = VLV_POSITION_OPENED;
782:          // queue up the next stage of the calibration
```

```
783:          push_time_queue_ISR(global_rtc_time + 3, CALIBRATE_VALVE_2);
784:       }
785:       // valve closing towards closing endstop pre-calibration
786:       else if (global_system_state == SYSTEM_CAL_VLV_2)
787:       {
788:          global_valve_position = VLV_POSITION_CLOSED;
789:          // queue up the next stage of the calibration
790:          push_time_queue_ISR(global_rtc_time + 3, CALIBRATE_VALVE_3);
791:       }
792:       // valve opening fully from fully closed for calibration
793:       else if (global_system_state == SYSTEM_CAL_VLV_3)
794:       {
795:          // set the valve position to open
796:          global_valve_position = VLV_POSITION_OPENED;
797:          // recalculate the valve opening time for each 1024th
798:          //    (valve movements (1024th) * the time it takes for each 1024th
799:          //    + the extra time) divided by 1024
800:          global_valve_time_to_open_1024th =                                    \
801:          ((((uint32_t) global_valve_time_in_motion_1024ths * (uint32_t) global_valve_time_to_open_1024th) \
802:          + temp16_frac) >> 10);
803:          // queue up the next stage of the calibration
804:          push_time_queue_ISR(global_rtc_time + 3, CALIBRATE_VALVE_5);
805:       }
806:       // valve closing fully from fully open for calibration
807:       else if (global_system_state == SYSTEM_CAL_VLV_5)
808:       {
809:          global_valve_position = VLV_POSITION_CLOSED;
810:          // recalculate the valve closing time for each 1024th
811:          //    (valve movements (1024th) * the time it takes for each 1024th
812:          //    + the extra time) divided by 1024
813:          global_valve_time_to_close_1024th =                                   \
814:          ((((uint32_t) global_valve_time_in_motion_1024ths * (uint32_t) global_valve_time_to_close_1024th) \
815:          + temp16_frac) >> 10);
816:          // queue up the next stage of the calibration
817:          push_time_queue_ISR(global_rtc_time + 3, CALIBRATE_VALVE_6);
818:       }
819:       // we are doing a valve endstop detect that we have a known calibration for.
820:       // We want to check the positional error (if we are too far away from the
821:       // endstop in position when the endstop is detected).  This applies to
822:       // normal moves as well as re-calibration moves.
823:       else
824:       {
825:          // valve has closed/opened fully in preparation of a recalibration of
826:          //    valve position.  Trigger a move valve to move to the pending valve
827:          //    set position and set system status to run
828:          if (global_system_state == SYSTEM_RECAL_VLV_MOVES)
```

```
829:          {
830:             push_time_queue_ISR(global_rtc_time + 3, MOVE_VALVE);
831:             global_system_state = SYSTEM_RUN;
832:          }
833:          // valve is closing
834:          if (IS_VLV_CLOSING)
835:          {
836:             // error checking if valve movement was longer or shorter than expected
837:             // We accomplish this by seeing if the valve hit an endstop while it was
838:             //    outside the VLV_NEAR_CLOSED_RANGE_MAX/VLV_NEAR_OPENED_RANGE_MAX
839:             if (global_valve_position > VLV_NEAR_CLOSED_RANGE_MAX)
840:             {
841:                // put system into run mode (in case it's in valve cal routine)
842:                global_system_state = SYSTEM_RUN;
843:                // set valve to unknown position (uncalibrated)
844:                global_valve_position = VLV_POSITION_UNKNOWN;
845:                // set the error bitfield and send an error
846:                global_error_message_bitfield |= ERR_MSG_VLV_MOVE_FAIL;
847:                push_time_queue_ISR(global_rtc_time + 1, MSG_MOTE_ERROR_MSG);
848:             }
849:             // was an expected endstop, proceed as usual
850:             else
851:             {
852:                global_valve_position = VLV_POSITION_CLOSED;
853:             }
854:          }
855:          // valve is opening
856:          else if (IS_VLV_OPENING)
857:          {
858:             // error checking if valve movement was longer or shorter than expected
859:             // We accomplish this by seeing if the valve hit an endstop while it was
860:             //    outside the VLV_NEAR_CLOSED_RANGE_MAX/VLV_NEAR_OPENED_RANGE_MAX
861:             if (global_valve_position < VLV_NEAR_OPENED_RANGE_MAX)
862:             {
863:                // put system into run mode (in case it's in valve cal routine)
864:                global_system_state = SYSTEM_RUN;
865:                // set valve to unknown position (uncalibrated)
866:                global_valve_position = VLV_POSITION_UNKNOWN;
867:                // set the error bitfield and send an error
868:                global_error_message_bitfield |= ERR_MSG_VLV_MOVE_FAIL;
869:                push_time_queue_ISR(global_rtc_time + 1, MSG_MOTE_ERROR_MSG);
870:             }
871:             // was an expected endstop, proceed as usual
872:             else
873:             {
874:                global_valve_position = VLV_POSITION_OPENED;
```

```
875:            }
876:          }
877:       }
878:
879:       // Update fixture setting
880:
881:
882:       // Turn off comparator and dac
883:       setup_DAC(DAC_OFF);
884:       setup_comparator(NC_NC_NC_NC);
885:
886:       // Turn off CCP3 interrupt
887:       disable_interrupts(INT_CCP3);
888:
889:       // reset valve movements counter
890:       global_valve_movements_since_endstop = 0;
891:
892:       // turn off the motor
893:       mV_COASTm;
894:
895:       // put clock speed down to 250khz again
896:       //fosc_250k_ISR();
897:
898:       // turn off comparator interrupt so it isn't triggered on stop
899:       clear_interrupt(INT_COMP);
900:    }
901: }
902:
903:
904: /*
905: #INT_TIMER0
906: void tmr0_ISR(void)
907: {
908: // may not be enabled during normal operation.
909: // Need only for IDLING situation (long winters...) wherein every 35minutes
910: //  we wake up and check the battery and set a flag if there is a very low
911: //  battery, which then tells the idle_sleep routine to go into deep_sleep.
912: // this needs work.
913: //
914: // We can know if we were idling by simply checking the OSCCON for 31250Hz
915: //  operation, which occurs ONLY during the long winter....
916: //
917:    if (cur_state == DEV_IDLE)
918:    {
919:    uint8_t  nn;
920:    // this interrupt occurred during the sleep_idle state
```

```
921:    // check the battery, if it is so low that we need to turn off the radio,
922:    //  then be sure to set a flag to make that happen...
923:    // Deep_Sleep may be the result
924:       output_high(AUX_PWR);
925:       ADON = TRUE;          // turn on ADC
926:       delay_cycles(50);     // insurance
927:       set_adc_channel(V_MEAS_REF);
928:       VfvrAD = 0;
929:       for (nn=0; nn<4; nn++)
930:          VfvrAD += read_adc();
931:       ADON = FALSE;
932:       output_low(AUX_PWR);
933:       if (VfvrAD > FVR_NODUST)
934:          fl_batNODUST = TRUE;
935:    }
936:    else
937:    {
938:    // any other TIMER0 activities we may want
939:    }
940: }
941: */
942:
943:
944: #INT_CCP3 HIGH
945: void ccp3_ISR(void)
946: {
947: // For use in timing valve motion
948: //    -Updates realtime position of valve
949: //    -Keeps track of valve movement time (in 1024th of full scale)
950: //    -Turns off valve movement when position is reached
951: //    -updates DAC level according to position and direction of movement
952: // May be used for other functionality if valve is not moving
953: // TMR1 dedicated to CCP3
954:
955:    //set_timer1(TIMER1_VLV_MOVE_INIT + get_timer1());
956:    //temp16_frac = get_timer1();
957:
958:    // If the valve is in motion (not in the braked or coast mode)
959:    if (IS_VLV_CLOSING || IS_VLV_OPENING)
960:    {
961:       // add to the time in motion variable
962:       global_valve_time_in_motion_1024ths++;
963:
964:       //check for valve movement timeout
965:       if (global_valve_time_in_motion_1024ths > VALVE_TIMEOUT)
966:       {
```

```
967:          // turn off comparator interrupt so it isn't triggered on stop
968:          disable_interrupts(INT_COMP);
969:
970:          // turn off valve movement
971:          mV_COASTm;
972:
973:          // Error handling for valve motion timeout
974:          // put system into run mode (in case it's in valve cal routine)
975:          global_system_state = SYSTEM_RUN;
976:          // set valve to unknown position (uncalibrated)
977:          global_valve_position = VLV_POSITION_UNKNOWN;
978:          // set the error bitfield and send an error
979:          global_error_message_bitfield |= ERR_MSG_VLV_MOVE_FAIL;
980:          push_time_queue_ISR(global_rtc_time + 1, MSG_MOTE_ERROR_MSG);
981:       }
982:
983:       else if (IS_VLV_CLOSING)
984:       {
985:          //set_timer1(TIMER1_VLV_MOVE_INIT);
986:          // 0x20 is equal to one 'millispan'
987:          if (global_valve_position >= 0x20)
988:          {
989:             global_valve_position -= 0x20;
990:          }
991:          // valve is closed, unsigned int thing
992:          else
993:          {
994:             global_valve_position = VLV_POSITION_CLOSED;
995:          }
996:
997:          // if system is doing a valve recalibration, ignore valve set position
998:          //    as it is storing the next valve position to move to
999:          if (global_system_state == SYSTEM_RECAL_VLV_MOVES)
1000:         {
1001:            break;
1002:         }
1003:         // we have reached or exceeded the set value/target position and
1004:         //    we aren't trying to reach the endstop
1005:         else if ((global_valve_position <= global_valve_position_set_value)&&\
1006:         (global_valve_position_set_value != VLV_POSITION_CLOSED))
1007:         {
1008:            // turn off comparator interrupt so it isn't triggered on stop
1009:            disable_interrupts(INT_COMP);
1010:
1011:            // turn off valve movement
1012:            mV_COASTm;
```

```
1013:          }
1014:       }
1015:       else if (IS_VLV_OPENING)
1016:       {
1017:          //0x20 is equal to one 'millispan'
1018:          global_valve_position += 0x20;
1019:
1020:          if (global_valve_position > VLV_POSITION_OPENED)
1021:          {
1022:             global_valve_position = VLV_POSITION_OPENED;
1023:          }
1024:
1025:          // if system is doing a valve recalibration, ignore valve set position
1026:          //    as it is storing the next valve position to move to
1027:          if (global_system_state == SYSTEM_RECAL_VLV_MOVES)
1028:          {
1029:             break;
1030:          }
1031:          // we have reached or exceeded the set value/target position and
1032:          //    we aren't trying to reach the endstop
1033:          else if ((global_valve_position >= global_valve_position_set_value)&&\
1034:          (global_valve_position_set_value != VLV_POSITION_OPENED))
1035:          {
1036:             // turn off comparator interrupt so it isn't triggered on stop
1037:             disable_interrupts(INT_COMP);
1038:
1039:             // turn off valve movement
1040:             mV_COASTm;
1041:
1042:             // Special case: if we are opening during CALIBRATE_VALVE_1 and
1043:             //    have reached our position, start the next calibration
1044:             //    sequence
1045:             if (global_system_state == SYSTEM_CAL_VLV_1)
1046:             {
1047:                push_time_queue_ISR(global_rtc_time + 3, CALIBRATE_VALVE_2);
1048:             }
1049:          }
1050:       }
1051:       // update the dac setting
1052:       set_comp_dac_level_isr();
1053:
1054:       // update the fixture setting
1055:
1056:       // if the valve is not moving anymore
1057:       if (IS_VLV_COASTING)
1058:       {
```

```
1059:          // switch back to lower clock, turn off CCP3 interrupt
1060:             //fosc_250k_ISR();
1061:             //#use delay(clock=250KHZ)
1062:          disable_interrupts(INT_CCP3);
1063:
1064:       }
1065:    }
1066: }
1067:
1068:
1069: #INT_CCP4
1070: void ccp4_isr(void)
1071: {
1072: // real time clock interrupts
1073: // TMR3 dedicated to CCP4
1074:
1075:    // increment global system uptime
1076:    global_rtc_time++;
1077:    // increment utc time if mote is connected
1078:    if (global_dust_enabled) global_utc_time++;
1079:
1080:    // check if a time queue item needs to be run
1081:    // make sure it isn't polling an empty queue
1082:    while((global_current_time_queue_location != 255) &&                        \
1083:    (global_current_priority_queue_location != (MAX_PRIORITY_QUEUE_ITEMS - 1)) \
1084:    && (global_time_queue[global_current_time_queue_location].time_to_execute  \
1085:    <= global_rtc_time))
1086:    {
1087:       // pop an item off the time queue and push it into the priority queue
1088:       pop_time_queue_ISR();
1089:    }
1090:
1091:    // check if a message queue item needs to be run
1092:    // make sure it isn't polling an empty queue
1093:    while((global_current_message_queue_location != 255) &&                      \
1094:       (global_message_queue[global_current_message_queue_location].time_to_send \
1095:       <= global_rtc_time))
1096:    {
1097:       // disable mote interrupt so payload_buf doesn't get overwritten
1098:       disable_interrupts(INT_EXT2_H2L);
1099:       // send message and requeue it at a later date if not ack'd
1100:       pop_message_queue_and_send_ISR();
1101:       // re-enable mote interrupt
1102:       enable_interrupts(INT_EXT2_H2L);
1103:    }
1104:
```

```
1105:
1106:    // check if a sprinkler queue item needs to be run
1107:    // make sure it isn't polling an empty queue
1108:    while((global_current_sprinkler_queue_location != 255) &&                        \
1109:       (global_sprinkler_queue[global_current_sprinkler_queue_location].start_time \
1110:       <= global_utc_time))
1111:    {
1112:             // if system is not in run state, do not stop the item.  Send an error
1113:       //    stating that the stop time is delayed
1114:       if (global_system_state != SYSTEM_RUN)
1115:       {
1116:          global_error_message_bitfield |= ERR_MSG_SPINKLER_CMD_DELAYED_INVALID_STATE;
1117:          push_time_queue_ISR(global_rtc_time + 1, MSG_MOTE_ERROR_MSG);
1118:       }
1119:       else
1120:       {
1121:          // pop an item off the time queue and push it into the priority queue
1122:          pop_sprinkler_queue_ISR();
1123:       }
1124:    }
1125:
1126:    // check if a sprinkler queue item needs to be stopped
1127:    if (global_current_sprinkler_settings_end_time <= global_utc_time)
1128:    {
1129:       // if system is not in run state, do not stop the item.  Send an error
1130:       //    stating that the stop time is delayed
1131:       if (global_system_state != SYSTEM_RUN)
1132:       {
1133:          global_error_message_bitfield |= ERR_MSG_SPINKLER_CMD_DELAYED_INVALID_STATE;
1134:          push_time_queue_ISR(global_rtc_time + 1, MSG_MOTE_ERROR_MSG);
1135:       }
1136:       else
1137:       {
1138:          // stop the current sprinkler setting
1139:          stop_current_spinkler_setting_ISR();
1140:       }
1141:    }
1142:
1143:    // update lcd every second
1144:    PUSH_PRIORITY_QUEUE_ISR_MACRO(LCD_UPDATE);
1145:
1146:    // periodic system checkups (all in one to minimize divides)
1147:    // - mote state (reacts as necessary)
1148:    // - battery state (turns off/on charging, sends warnings, etc.)
1149:    // - mote temp (logs data, sends warnings, etc.)
1150:    // - mote utc time (updates utc time if valid)
```

```
1151:    if ((global_rtc_time % PERIODIC_CHECKS_TIME) == 0)
1152:    {
1153:       PUSH_PRIORITY_QUEUE_ISR_MACRO(CHECK_MOTE_STATE);
1154:       PUSH_PRIORITY_QUEUE_ISR_MACRO(CHECK_BATTERY_STATE);
1155:       PUSH_PRIORITY_QUEUE_ISR_MACRO(CHECK_MOTE_TEMP);
1156:       PUSH_PRIORITY_QUEUE_ISR_MACRO(UPDATE_MOTE_TIME);
1157:    }
1158: }
1159:
1160: #INT_CCP5 FAST
1161: void CCP5_ISR(void)
1162: {
1163: // GEN_RPM event capture for determining speed of rotation
1164: // We need the CCP5 interrupt routine to be very fast because
1165: //  the sprinkler can be spinning fast enough to generate 500 pulses per sec!
1166: // With a 250KHz system clock, 2ms may be trouble with the full normal interrupt
1167: //  overhead....
1168: // TMR5 dedicated to CCP5
1169:
1170:    static uint16_t ccp5_value = 0, ccp5_value_prev = 0, previous_current_period = 0;
1171:
1172:    ccp5_value_prev = ccp5_value;             // save previous sample
1173:    ccp5_value = CCP_5;                       // get current sample
1174:    previous_current_period = global_current_period;                  // save previous difference
1175:    global_current_period = ccp5_value - ccp5_value_prev;   // calc current difference
1176:    global_last_rpm_value_time = global_rtc_time;
1177:    // INSERT code to ensure that cur_PER is 'legit', if needed
1178:
1179:    //  probably do not need this interrupt ctr functionality
1180:    // ccp5_intctr++;
1181:    // if (bit_test(ccp5_intctr++,5))
1182:    //   fl_loop_done = TRUE;     // this flag is cleared in main at top of loop
1183:
1184: }
1185:
```