```
 1 /*
 2  * Nickolas Schiffer #012279319
 3  * CMPE 146 S19
 4  * Watchdog Lab
 5  */
 6
 7 /**
 8  * @file
 9  * @brief This is the application entry point.
10  */
11
12 #include <stdio.h>
13 #include "io.hpp"
14 #include <tasks.hpp>
15 #include <storage.hpp>
16 #include <string>
17 #include <event_groups.h>
18 #include <ff.h>
19 #include <cstring>
20 #include <task.h>
21 #include <str.hpp>
22
23 #define PRODUCER_BIT (1 << 1)
24 #define CONSUMER_BIT (1 << 2)
25
26 typedef struct {
27     uint16_t avg;
28     uint64_t time;
29 } light_entry_t;
30
31 typedef struct {
32     QueueHandle_t *queue;
33     EventGroupHandle_t *event_group;
34
35 } task_params_t;
36
37 QueueHandle_t sensor_queue = xQueueCreate(100, sizeof(light_entry_t));
38 EventGroupHandle_t event_group = xEventGroupCreate();
39
40 TaskHandle_t producer_handle = NULL, consumer_handle = NULL, watchdog_handle = NULL,
   cpu_info_handle = NULL;
41
42 CMD_HANDLER_FUNC(taskHandler)
43 {
44 //    uint8_t numtasks = 0;
106     auto params = (char *) cmdParams.c_str();
107     printf("command: %s\n", params);
108     char *pch = NULL;
109     uint8_t numtasks = 0;
110
111     pch = strtok(params, " ");
112
113     if (strcasecmp(pch, "suspend") == 0) {
114         pch = strtok(NULL, " ");
115         while (pch != NULL) {
116             if (strcasecmp(pch, "consumer") == 0) {
117                 vTaskSuspend(consumer_handle);
```

```
118                 puts("Suspended Consumer");
119                 numtasks++;
120             }
121             else if (strcasecmp(pch, "producer") == 0) {
122                 vTaskSuspend(producer_handle);
123                 puts("Suspended Producer");
124                 numtasks++;
125             }
126             else if (strcasecmp(pch, "watchdog") == 0) {
127                 vTaskSuspend(watchdog_handle);
128                 puts("Suspended Watchdog");
129                 numtasks++;
130             }
131             else {
132                 printf("\"%s\" is not a valid task.\n", pch);
133             }
134             pch = strtok(NULL, " ");
135         }
136         if (numtasks) {
137             printf("%d tasks suspended.\n", numtasks);
138             return true;
139         }
140         else {
141             return false;
142         }
143     }
144     else if (strcasecmp(pch, "resume") == 0) {
145         puts("resuming");
146         pch = strtok(NULL, " ");
147         while (pch != NULL) {
148             if (strcasecmp(pch, "consumer") == 0) {
149                 vTaskResume(consumer_handle);
150                 puts("Resumed Consumer");
151                 numtasks++;
152             }
153             else if (strcasecmp(pch, "producer") == 0) {
154                 vTaskResume(producer_handle);
155                 puts("Resumed Producer");
156                 numtasks++;
157             }
158             else if (strcasecmp(pch, "watchdog") == 0) {
159                 vTaskResume(watchdog_handle);
160                 puts("Resumed Watchdog");
161                 numtasks++;
162             }
163             else {
164                 printf("\"%s\" is not a valid task.\n", pch);
165             }
166             pch = strtok(NULL, " ");
167         }
168         if (numtasks) {
169             printf("%d tasks resumed.\n", numtasks);
170             return true;
171         }
172         else {
173             return false;
174         }
```

```
175     }
176
177 //    while (pch != NULL)
182
183 //    auto c = cmdParams.getToken(" ", false);
224
225     return true;
226 }
227
228 void vTaskGetHandle(void *pvParamters)
229 {
230     std::string handle_name = "producer";
231     TaskHandle_t handle1 = NULL, handle2 = NULL, handle3 = NULL, handle4 = NULL;
232     while (1) {
233         handle1 = xTaskGetHandle("producer");
234         handle2 = xTaskGetHandle("consumer");
235         handle3 = xTaskGetHandle("get_handle");
236         handle4 = xTaskGetIdleTaskHandle();
237         printf("%X\n%X\n%X\n%X\n\n", handle1, handle2, handle3, handle4);
238         vTaskDelay(1000);
239     }
240 }
241
242 void vTokenizer(void *pvParameters)
243 {
244
245     while (1) {
246         str string = "Hello my name is Simon";
247         str *token = nullptr;
248
249         while (1) {
250             auto t = string.getToken(" ", false);
251             if (t != NULL) {
252                 puts(t->c_str());
253             }
254             else {
255                 break;
256             }
257
258         }
259         vTaskDelay(100);
260     }
261 }
262
263 void vCPUUsageTask(void *pvParameters)
264 {
265     std::string file = "";
266     char buffer[BUFSIZ + 1];
267
268     while (1) {
269         const int delayInMs = 60000;
270
271         snprintf(buffer, BUFSIZ, "CPU Info Logging started at %lums\n", xTaskGetMsCount());
272         file += buffer;
273
274         if (delayInMs > 0) {
275             vTaskResetRunTimeStats();
```

```
276             vTaskDelayMs(delayInMs);
277         }
278 
279         // Enum to char : eRunning, eReady, eBlocked, eSuspended, eDeleted
280         const char * const taskStatusTbl[] = { "RUN", "RDY", "BLK", "SUS", "DEL" };
281 
282         // Limit the tasks to avoid heap allocation.
283         const unsigned portBASE_TYPE maxTasks = 16;
284         TaskStatus_t status[maxTasks];
285         uint32_t totalRunTime = 0;
286         uint32_t tasksRunTime = 0;
287         const unsigned portBASE_TYPE uxArraySize = uxTaskGetSystemState(&status[0], maxTasks,
    &totalRunTime);
288         snprintf(buffer, BUFSIZ, "%10s Sta Pr Stack CPU%%          Time\n", "Name");
289         file += buffer;
290         for (unsigned priorityNum = 0; priorityNum < configMAX_PRIORITIES; priorityNum++) {
291             /* Print in sorted priority order */
292             for (unsigned i = 0; i < uxArraySize; i++) {
293                 TaskStatus_t *e = &status[i];
294                 if (e->uxBasePriority == priorityNum) {
295                     tasksRunTime += e->ulRunTimeCounter;
296 
297                     const uint32_t cpuPercent = (0 == totalRunTime) ? 0 : e->ulRunTimeCounter
    / (totalRunTime / 100);
298                     const uint32_t timeUs = e->ulRunTimeCounter;
299                     const uint32_t stackInBytes = (4 * e->usStackHighWaterMark);
300 
301                     snprintf(buffer, BUFSIZ, "%10s %s %2u %5u %4u %10u us\n", e->pcTaskName,
    taskStatusTbl[e->eCurrentState], e->uxBasePriority, stackInBytes,
302                             cpuPercent, timeUs);
303                     file += buffer;
304                 }
305             }
306         }
307 
308         /* Overhead is the time not accounted towards any of the tasks.
309          * For example, when an ISR happens, that is not part of a task's CPU usage.
310          */
311         const uint32_t overheadUs = (totalRunTime - tasksRunTime);
312         const uint32_t overheadPercent = overheadUs / (totalRunTime / 100);
313         snprintf(buffer, BUFSIZ, "%10s --- -- ----- %4u %10u luS\n", "(overhead)",
    overheadPercent, overheadUs);
314         file += buffer;
315         if (uxTaskGetNumberOfTasks() > maxTasks) {
316             snprintf(buffer, BUFSIZ, "** WARNING: Only reported first %lu tasks\n", maxTasks);
317             file += buffer;
318         }
319         file += "\n\n\n";
320         Storage::append("1:cpu.txt", file.c_str(), file.size());
321         file.clear();
322         memset(buffer, 0, sizeof(buffer));
323         puts("CPU Info logged to cpu.txt");
324 
325     }
326 }
327 
328 void vProducerTask(void *pvParameters)
```

```
329 {
330
331     Light_Sensor light_sensor = Light_Sensor::getInstance();
332     light_sensor.init();
333     uint8_t counter = 0;
334     uint32_t sum = 0;
335     light_entry_t entry = { 0, 0 };
336     uint64_t start_time = xTaskGetMsCount();
337
338     while (1) {
339
340         if (counter == 99) {
341             entry.avg = (uint16_t) sum / 100;
342             entry.time = (uint64_t) xTaskGetMsCount() - start_time;
343             //printf("%lu, %lu\n", entry.time, entry.avg);
344             //puts("producer checking in");
345             xQueueSend(sensor_queue, &entry, portMAX_DELAY);
346             sum = counter = 0;
347         }
348         else {
349             sum += light_sensor.getRawValue();
350             counter++;
351             vTaskDelay(1);
352         }
353         xEventGroupSetBits(event_group, PRODUCER_BIT);
354
355     }
356
357 }
358
359 void vConsumerTask(void *pvParameters)
360 {
361     light_entry_t entry = { 0, 0 };
362     uint8_t counter = 0;
363     std::string buffer = "";
364
365     char string[BUFSIZ + 1] = { 0 };
366
367     while (1) {
368         xQueueReceive(sensor_queue, &entry, portMAX_DELAY);
369         snprintf(&string[0], BUFSIZ, "%lu, %u\n", (unsigned long) entry.time, entry.avg);
370         //printf("%lu, %u\n", (unsigned long)entry.time, entry.avg);
371         buffer += string;
372         counter++;
373         if (counter == 10) {
374             counter = 0;
375             Storage::append("1:test.txt", buffer.c_str(), buffer.size());
376             buffer.clear();
377         }
378         xEventGroupSetBits(event_group, CONSUMER_BIT);
379
380     }
381 }
382
383 void vWatchDogTask(void *pvParameters)
384 {
385     std::string buffer = "";
```

```
386     char string[BUFSIZ + 1] = { 0 };
387     while (1) {
388         xEventGroupClearBits(event_group, (PRODUCER_BIT | CONSUMER_BIT));
389         EventBits_t bits = xEventGroupWaitBits(event_group, (PRODUCER_BIT | CONSUMER_BIT),
390         pdTRUE,
391         pdTRUE, 1000);
392         //printf("bits: %X\n", bits);
393         if (!(bits & (PRODUCER_BIT))) {
394             if ((!(bits & (CONSUMER_BIT)))) {
395                 //both bits unset
396                 snprintf(&string[0], BUFSIZ, "time: %lu: Both Tasks are Stuck.\n",
    xTaskGetMsCount());
397                 buffer += string;
398                 Storage::append("1:stuck.txt", buffer.c_str(), buffer.size());
399                 memset(string, 0, sizeof(string));
400                 buffer.clear();
401                 //puts("Producer and Consumer Stuck");
402             }
403             else {
404                 //just producer unset
405                 snprintf(&string[0], BUFSIZ, "time: %lu: Producer Task is Stuck.\n",
    xTaskGetMsCount());
406                 buffer += string;
407                 Storage::append("1:stuck.txt", buffer.c_str(), buffer.size());
408                 memset(string, 0, sizeof(string));
409                 buffer.clear();
410                 //puts("Producer Stuck");
411             }
412         }
413         else if (!(bits & (CONSUMER_BIT))) {
414             //just consumer unset
415             snprintf(&string[0], BUFSIZ, "time: %lu: Consumer Task is Stuck.\n",
    xTaskGetMsCount());
416             buffer += string;
417             Storage::append("1:stuck.txt", buffer.c_str(), buffer.size());
418             memset(string, 0, sizeof(string));
419             buffer.clear();
420             //puts("Consumer Stuck");
421         }
422 
423     }
424 }
425 
426 int main(void)
427 {
428 
429     scheduler_add_task(new terminalTask(PRIORITY_HIGH));
430 
431     f_unlink("1:test.txt");
432     f_unlink("1:stuck.txt");
433     f_unlink("1:cpu.txt");
434 
435     xTaskCreate(vProducerTask, "producer", 1024, NULL, PRIORITY_MEDIUM, &producer_handle);
436     xTaskCreate(vConsumerTask, "consumer", 1024, NULL, PRIORITY_MEDIUM, &consumer_handle);
437     xTaskCreate(vWatchDogTask, "watchdog", 1024, NULL, PRIORITY_HIGH, &watchdog_handle);
438     xTaskCreate(vCPUUsageTask, "CPU Usage", 1024, NULL, PRIORITY_MEDIUM, &cpu_info_handle);
439     //xTaskCreate(vTokenizer, "Tokenizer", 1024, NULL, PRIORITY_LOW, NULL);
```

```
440     //xTaskCreate(vTaskGetHandle, "get_handle", 1024, NULL, PRIORITY_LOW, NULL);
441
442     scheduler_start();
443     return -1;
444 }
445
```

```
69 bool terminalTask::taskEntry()
70 {
71     /* remoteTask() creates shared object in its init(), so we can get it now */
72         CommandProcessor &cp = mCmdProc;
73 
74     /* Task Command for suspending/resuming tasks (Watchdogs Lab) */
75     CMD_HANDLER_FUNC(taskHandler);
76     cp.addHandler(taskHandler, "task", "Suspend or Resume a Task by name. \"task suspend task1\"
   should suspend a task named \"task1\" ");
```