```c
#include "_kernelCore.h"
#include "osDefs.h"

//global variables
extern threadStruct threadCollection[MAX_THREADS];
extern int numThreads;
int threadCurr = 0;
extern int idleIndex;
bool leaveIdle = false;
bool canInterrupt = false;


//set priority of the PendSV interrupt
void kernelInit(void){
  SHPR3 |= 0xFF << 16;
}

//start running the kernel, i.e. the OS
bool osKernelStart(){
  threadCurr = 0;
  if(numThreads > 0)
  {
    __set_CONTROL(1<<1); //enter threading
    __set_PSP((uint32_t) threadCollection[threadCurr].TSP); //set PSP to the first thread address

    osLoadFirst(); //begin running threads
  }

  return false; //once called, function should not end unless something went wrong in OS
}

//start running the first thread, which will lead into context switching between all the threads
void osLoadFirst(){
    ICSR |= 1<<28;
    __asm("isb");
}

//called when a thread yields, starts task switching process
void osYield(void){
  canInterrupt = false;

  //move TSP of the running thread 16 memory locations lower, so that next time the thread loads the 16 context registers, we end at the same PSP
  threadCollection[threadCurr].TSP = (uint32_t*)(__get_PSP()-16*4);
  //if the thread is able to sleep, set it to sleep
  if(threadCollection[threadCurr].sleepTime != 0){
    threadCollection[threadCurr].status = SLEEPING;
    threadCollection[threadCurr].timer = threadCollection[threadCurr].sleepTime; //set timer to user-defined sleep timer
  }
  //otherwise, set it back to waiting
  else{
    threadCollection[threadCurr].status = WAITING;
  }

  scheduler();

  canInterrupt = true;
  ICSR |= 1<<28;
  __asm("isb");
}

//determine next available thread to switch to
void scheduler(void){
  bool isFound = false;
  int index = threadCurr;
  if(threadCurr == idleIndex){
    index = threadCurr-1; //want to cycle through the valid thread indexes
  }

  if (numThreads > 1){
    for (int i = 0; i < numThreads && isFound == false; i++){
```

```
71          //cycle through the threads in the thread struct array
72          index = (index+1)%numThreads;
73          printf("Trying thread %d\n", index+1);
74
75          //check status: if next thread in round robin is waiting, proceed! else, loop back and look at
    next thread
76          if(threadCollection[index].status == WAITING){
77            threadCurr = index;
78            isFound = true;
79          }
80        }
81
82        //if no threads are waiting, use idle thread
83        if(isFound != true){
84          printf("all the people of the world are asleep\n");
85          threadCurr = idleIndex;
86        }
87      }
88    }
89
90    void SysTick_Handler(void){
91      //decrement running timeslice
92      --threadCollection[threadCurr].timer;
93      //printf("thread num %d, timer: %d\n", threadCurr+1, threadCollection[threadCurr].timer);
94
95      //decrement sleep timers
96      for(int i = 0; i < numThreads; i++){
97        if(threadCollection[i].status == SLEEPING && i != threadCurr)
98        {
99          --threadCollection[i].timer;
100         /*if(threadCollection[i].timer % 50 == 0){
101           printf("Thread %d sleeptime: %d \n", (i+1), threadCollection[i].timer);
102         }*/
103
104         //check wake-up status
105         if(threadCollection[i].timer <= 0)
106         {
107           if(threadCurr == idleIndex){
108             leaveIdle = true;
109           }
110           threadCollection[i].status = WAITING;
111           threadCollection[i].timer = threadCollection[i].timeslice;
112         }
113       }
114     }
115
116     //if timeslice of running thread is up, proceed with task-switching
117     if((threadCollection[threadCurr].timer <= 0 || leaveIdle == true) && canInterrupt == true)
118     {
119       printf("Thread timer complete\n");
120       threadCollection[threadCurr].TSP = (uint32_t*)(__get_PSP()-8*4); //decrement PSP only 8 locations
    lower, since the hardware registers remain on the stack
121       //prepare current thread to sleep if can sleep
122       if(threadCollection[threadCurr].sleepTime != 0){
123         threadCollection[threadCurr].status = SLEEPING;
124         threadCollection[threadCurr].timer = threadCollection[threadCurr].sleepTime;
125       }
126       //if thread doesn't sleep, set status to waiting and timer to timeslice
127       else{
128         threadCollection[threadCurr].status = WAITING;
129         threadCollection[threadCurr].timer = threadCollection[threadCurr].timeslice;
130       }
131       //if a thread woke up, return to round robin
132       if(leaveIdle == true){
133         threadCurr = idleIndex-1;
134         leaveIdle = false;
135       }
136
137       scheduler();
138
139       ICSR |= 1<<28;
140       __asm("isb");
```

```
141     }
142   }
143
144
145   int task_switch(void){
146     //set PSP to the thread we want to start running
147     __set_PSP((uint32_t)threadCollection[threadCurr].TSP);
148     threadCollection[threadCurr].status = ACTIVE;
149
150     if (threadCurr == numThreads ){
151       printf("Running idle thread \n");
152     }
153     return 0;
154   }
155
```