```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;


//set priority of the PendSV interrupt
void kernelInit(void){
  //PendSV priority
  SHPR3 |= 0xFE << 16;
  SHPR3 |= 0xFFU << 24; //SysTick priority

  SHPR2 |= 0xFDU << 24; //SVC priority
}

//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(){
  if(numThreads < 1){
    threadCurr = idleIndex;
  }

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

//called when a non-periodic thread is set to sleep, starts task switching process
void osSleep(int sleepTime){
  threadCollection[threadCurr].status = SLEEPING;
  threadCollection[threadCurr].timer = sleepTime;

  printf("Thread yielded from osSleep.\n");
  __ASM("SVC #0");
}

//called when a thread yields, starts task switching process
void osYield(void){
  printf("Thread yielded from osYield.\n");
  __ASM("SVC #0");
}

//determine next available thread to switch to
void scheduler(void){
  bool isFound = false;
  int shortestDeadIndex = 0;

  if(numThreads > 0){
    for (int i = 0; i < numThreads; i++)
    {
      printf("Time on thread %d: %d, Status: %d\n", i+1, threadCollection[i].timer,
threadCollection[i].status);

      //look for the earliest deadline among waiting tasks
      if(threadCollection[i].status == WAITING){
```

```
72          if(isFound == false){ //at least one waiting task found
73            isFound = true;
74            shortestDeadIndex = i;
75
76          }
77          if(isFound == true){ //find earlier deadlines than the one found
78            if(threadCollection[i].timer < threadCollection[shortestDeadIndex].timer){ //if there's a
    tie, stick with the thread of the lower index
79              shortestDeadIndex = i;
80            }
81          }
82        }
83      }
84    }
85
86    //if no threads are waiting, use idle thread
87    if(isFound != true){
88      printf("all the people of the world are asleep\n");
89      threadCurr = idleIndex;
90    }
91    else{
92      threadCurr = shortestDeadIndex;
93      printf("Trying thread %d\n", threadCurr+1);
94    }
95  }
96
97  void SysTick_Handler(void){
98    bool preEmptTask = false;
99
100   for(int i = 0; i < numThreads; i++)
101   {
102     --threadCollection[i].timer; //decrement all timers: deadlines and sleep timers
103
104     if(threadCollection[i].timer <= 0) //i is within numThreads range, therefore does not include the
    idleThread
105     {
106       //thread failed to meet its deadline. user's fault for designing poor threads
107       if(threadCollection[i].status == WAITING){
108         printf("Deadline of thread %d missed -- system failed.\n", i+1);
109       }
110
111       //check wake-up status if sleeping
112       if(threadCollection[i].status == SLEEPING)
113       {
114         printf("Waking up thread %d\n", i+1);
115
116         if(threadCurr == idleIndex){
117             leaveIdle = true;
118         }
119         threadCollection[i].status = WAITING;
120         threadCollection[i].timer = threadCollection[i].deadline;
121
122         //if the thread that has woken up has an earlier deadline than the running task, or has a tied
    deadline
123           //if tied, choose the thread with the lower index
124         if (threadCollection[i].timer < threadCollection[threadCurr].timer ||
    (threadCollection[i].timer == threadCollection[threadCurr].timer && i < threadCurr)){
125           preEmptTask = true;
126         }
127       }
128     }
129   }
130
131   //leave current thread if preempted by another thread with a earlier deadline, have been signalled to
    leave the idle thread, or if the running thread's timer goes to zero
132   if(preEmptTask == true || leaveIdle == true || threadCollection[threadCurr].timer <= 0)
133   {
134     printf("Thread yielded from SysTick.\n");
135
136     threadCollection[threadCurr].TSP = (uint32_t*)(__get_PSP()-8*4); //decrement PSP only 8 locations
    lower, since the hardware registers remain on the stack
137
```

```
138       //set to sleep if periodic
139       if(threadCollection[threadCurr].period != 0){
140         threadCollection[threadCurr].status = SLEEPING;
141         threadCollection[threadCurr].timer = threadCollection[threadCurr].period;
142       }
143       else{
144         threadCollection[threadCurr].status = WAITING;
145         threadCollection[threadCurr].timer = threadCollection[threadCurr].deadline;
146       }
147
148       //reset leaveIdle flag
149       if(leaveIdle == true){
150         leaveIdle = false;
151       }
152
153       scheduler();
154
155       ICSR |= 1<<28;
156       __asm("isb");
157     }
158   }
159
160   void SVC_Handler_Main(uint32_t *svc_args)
161   {
162     char call = ((char*)svc_args[6])[-2];
163
164     if(call == 0) //code for a thread that has yielded/yielded by osSleep()
165     {
166       //move TSP of the running thread 8 memory locations lower, so that next time the thread loads the
    16 context registers, we end at the same PSP
167       threadCollection[threadCurr].TSP = (uint32_t*)(__get_PSP()-8*4);
168       //in the case of a periodic thread that yields
169       if(threadCollection[threadCurr].period != 0){
170         threadCollection[threadCurr].status = SLEEPING;
171         threadCollection[threadCurr].timer = threadCollection[threadCurr].period;
172       }
173       //if not periodic and if not yielded by osSleep
174       else if(threadCollection[threadCurr].status != SLEEPING){
175         threadCollection[threadCurr].status = WAITING;
176         threadCollection[threadCurr].timer = threadCollection[threadCurr].deadline;
177       }
178
179       scheduler();
180
181       ICSR |= 1<<28;
182       __asm("isb");
183     }
184   }
185
186
187   int task_switch(void){
188     //set PSP to the thread we want to start running
189     __set_PSP((uint32_t)threadCollection[threadCurr].TSP);
190     threadCollection[threadCurr].status = ACTIVE;
191
192     if (threadCurr == idleIndex ){
193       printf("Running idle thread \n");
194     }
195     return 0;
196   }
197
198
199
```