```c
1   #include "_threadsCore.h"
2   #include "osDefs.h"
3
4   //global variables
5   uint32_t* endOfStack_ptr = NULL;
6   int numThreads = 0;
7   int idleIndex = 0;
8   threadStruct threadCollection[MAX_THREADS];
9
10  //obtain the initial location of MSP by looking it up in the vector table
11  uint32_t* getMSPInitialLocation (void){
12    uint32_t* MSP_ptr = (uint32_t*) 0x0; //define a pointer to a pointer that points to initial MSP
13    printf("MSP: %08x\n", *MSP_ptr);
14    if(endOfStack_ptr == NULL){ //only allow endOfStack_ptr to be set to initial MSP location once
15      endOfStack_ptr = (uint32_t*) *MSP_ptr;
16    }
17
18    return (uint32_t*) *MSP_ptr; //dereference so that it returns just the pointer to initial MSP
19  }
20
21  //return address of new a PSP with offset of "offset" bytes from MSP
22  uint32_t* getNewThreadStack (uint32_t offset){
23    //check if we are exceeding the max stack size
24    if (MAX_STACK < offset){
25      printf("ERROR: Offset too large");
26      return NULL;
27        //make sure to look for a NULL return in future functions to check if getNewThreadStack failed or
    not
28    }
29
30    //calculate address of PSP from MSP
31    uint32_t* MSP_ptr = getMSPInitialLocation();
32    uint32_t PSP_adr = (uint32_t) MSP_ptr - offset;
33
34    //check if PSP address is a number divisible by 8
35    if(PSP_adr%8 != 0){
36      PSP_adr = PSP_adr+sizeof(uint32_t); //add 4 to address to ensure valid address for the stack
37    }
38
39    //check if overwriting a previous stack
40    if(PSP_adr > (uint32_t) endOfStack_ptr-(STACK_SIZE)){
41      printf("ERROR: Overwriting old data");
42      return NULL;
43    }
44
45    //assign PSP_ptr to point to PSP_adr
46    uint32_t* PSP_ptr = (uint32_t*) PSP_adr;
47    printf("PSP: %08x\n", (uint32_t) PSP_ptr);
48    endOfStack_ptr = PSP_ptr;
49
50    return PSP_ptr;
51  }
52
53
54  //LAB 1: set the value of PSP to threadStack and ensure that the microcontroller is using that value by
    changing the CONTROL register
55  /*void setThreadingWithPSP (uint32_t* threadStack){
56    __set_PSP((uint32_t) threadStack);
57    __set_CONTROL(1<<1);
58  }*/
59
60
61  //Initializes the thread stack and its initial context in memory
62  int osThreadNew(void (*fun_ptr)(void), int deadline, int period){
63    ++numThreads;
64    int stackID = numThreads-1;
65
66    //generate and store TSP
67    if(threadCollection[stackID].fun_ptr == idleThread){
68      threadCollection[stackID].TSP = getNewThreadStack(STACK_SIZE + numThreads*STACK_SIZE); //in the
    future, use a smaller stack size for idle & modify getNewThreadStack to allow for this
69    }
```

```c
 70       else{
 71         threadCollection[stackID].TSP = getNewThreadStack(STACK_SIZE + numThreads*STACK_SIZE); //MSP stack
      + n*thread stacks
 72       }
 73
 74      //if getnewThreadStack encounters an error creating the thread pointer, TSP generated will be a NULL
      pointer
 75      if(threadCollection[stackID].TSP == NULL){
 76        --numThreads;
 77        return -1; //osThreadNew failed
 78      }
 79
 80      //set threadStruct params
 81      threadCollection[stackID].fun_ptr = fun_ptr;
 82      threadCollection[stackID].timer = deadline;
 83      threadCollection[stackID].deadline = deadline;
 84      threadCollection[stackID].period = period;
 85
 86      //set the values for what the "running" thread will populate the registers with
 87      *(--threadCollection[stackID].TSP) = 1<<24; //xPSR
 88      *(--threadCollection[stackID].TSP) = (uint32_t) fun_ptr; //PC (program counter)
 89
 90        //dummy values (need to be nonzero)
 91        *(--threadCollection[stackID].TSP) = 0xE; //LR
 92        *(--threadCollection[stackID].TSP) = 0xC; //R12
 93        *(--threadCollection[stackID].TSP) = 0x3; //R3
 94        *(--threadCollection[stackID].TSP) = 0x2; //R2
 95        *(--threadCollection[stackID].TSP) = 0x1; //R1
 96        *(--threadCollection[stackID].TSP) = 0x0; //R0
 97
 98        //dummy values (for testing purposes)
 99        *(--threadCollection[stackID].TSP) = 0xB; //R11
100        *(--threadCollection[stackID].TSP) = 0xA; //R10
101        *(--threadCollection[stackID].TSP) = 0x9; //R9
102        *(--threadCollection[stackID].TSP) = 0x8; //R8
103        *(--threadCollection[stackID].TSP) = 0x7; //R7
104        *(--threadCollection[stackID].TSP) = 0x6; //R6
105        *(--threadCollection[stackID].TSP) = 0x5; //R5
106        *(--threadCollection[stackID].TSP) = 0x4; //R4
107
108      threadCollection[stackID].status = WAITING;
109
110      //if the idle thread is being initialized
111      if(threadCollection[stackID].fun_ptr == idleThread){
112        --numThreads;
113        idleIndex = numThreads;
114      }
115
116      return 0;
117    }
118
119    //idle function
120    void idleThread(void){
121      while(1);
122    }
```