***MANUAL***

In order to use the Multi-threading feature of this OS, you need to enable memory region that will be used to reside the thread control blocks(TCB).

To do that, follow the following steps,

* Open a text document.
* Copy and paste following line in the text file,

MAP 0x3FFFFB44,0x3FFFFFF8 read write

* Save the file as “STM.ini” in your project folder.
* Now open keil uVision4, and go to “options for target”
* Select “Debug tab”
* In the “initialization File” on the left, add this file.

0x3FFFFB44 to 0x3FFFFFF8 is a memory region that will store thread control blocks. By default, this memory region is disabled. We have to enable it using the steps mentioned above.

DO NOT WRITE TO THIS MEMORY REGION, ELSE YOU’LL CRASH THE OS.

After enabling the memory, the first thing you want to do is call the function Init() from main(). The Init() function is mandatory to call as it initializes the OS Timer. Without initializing the timer, there will be no scheduling.

int main(){

Init();

while(1)

{}

}

***MULTI-THREADING FUNCTIONS:***

1. ***CREATING A THREAD:***

In order to create a thread, the function is:

Create\_Task(void\* funcAddr)

And it takes the address of the function you want to run as a thread. That function must not have any return type or parameters. In other words, the return type and parameter of the function must be void.

Create\_Task() does not handle functions with parameters or return type. You can create up to 10 threads. The function that you want to run as a thread must contain an infinite loop. In other world the function must not be returned. The prototype of the function must include “\_\_attribute\_\_((noreturn))”, this tells the compiler that the function will not return. (see example).

The function returns 0 if unsuccessful, else it returns thread ID. You can create threads inside other thread functions also.

***Example:***

void Task\_1(void) \_\_attribute\_\_((noreturn)); // Prototype

void Task\_1(void){

while(1){ // Infinite loop is mandatory

// perform your task here

}

}

int main(){

Init();

int task\_id = Create\_Task(Task\_1); // Function’s name is its address

while(1){}

}

This small program will create a thread running Task\_1 function and that thread will get scheduled. Each thread has its own stack. The size of the stack is 120 bytes but 80 bytes are used by the OS and remaining 40 bytes are used for local variables. Keep this in mind when declaring variables in thread. You have 40 bytes for local thread variables. If you exceed stack size, it might cause undefined behavior.

1. ***Deleting A THREAD:***

In order to delete a thread, the function is:

Delete\_Task(int id)

And it takes the ID of the thread.

The function returns 0 if unsuccessful, and returns 1 on successful.

Thread can only be deleted in main() function(main process). Threads cannot be deleted in its own function, nor it can be deleted from other thread functions.

***Example:***

Continuing the previous example.

int main(){

Init();

int task\_id = Create\_Task(Task\_1); // Function’s name is its address

while(1){

if(SomeCodition)

Delete\_Task(task\_id);

}

}

***THREAD-SYNCHRONIZATION FUNCTIONS:***

Thread synchronization is the concurrent execution of two or more threads that share critical resources. However, conflicts may arise when parallel-running thread attempt to modify a common variable at the same time.

In order to avoid these conflicts, the OS provides mutex/semaphores functions.

***SEMAPHORES:***

* sem\_init(semaphore\* sem\_obj, int value) // returns 1 on success, else 0
* void sem\_wait(semaphore\* sem\_obj)
* void sem\_signal(semaphore\* sem\_obj)
* int sem\_getValue(semaphore\* sem\_obj) // returns current value of semaphore

***MUTEX:***

* void mutex\_init(mutex\* obj)
* void mutex\_lock(mutex\* obj)
* void mutex\_unlock(mutex\* obj)

This Example Demonstrates how to use **Mutex**:

mutex lock;

int shared\_variable = 0;

void Task\_1(void) \_\_attribute\_\_((noreturn);

void Task\_1(void){

while(1){

mutex\_lock(&lock);

--shared\_variable;

mutex\_unlock(&lock);

}

}

int main(){

Init();

mutex\_init(&lock);

int task\_id = Create\_Task(Task\_1);

while(1){

mutex\_lock(&lock);

++shared\_variable;

mutex\_unlock(&lock);

}

return 0;

}

Similarly, you can use semaphores in the same way. This example shows how to initialize and use semaphore variable.

semaphore sem\_lock;

int shared\_variable = 0;

void Task\_1(void) \_\_attribute\_\_((noreturn);

void Task\_1(void){

while(1){

sem\_wait(&sem\_lock);

--shared\_variable;

sem\_signal(&sem\_lock);

}

}

int main(){

Init();

sem\_init(&sem\_lock, 3) /\* 3 is the value of semaphore variable, you can use any other value according to your need but it must be less than 11.\*/

int task\_id = Create\_Task(Task\_1);

while(1){

sem\_wait(&sem\_lock);

++shared\_variable;

sem\_signal(&sem\_lock);

}

return 0;

}