MicroKernel

v1.1.0

Table of contents

Overview

MicroKernel is a simple kernel implementation for Cortex-M based microcontrollers. It provides concurrency and some synchronization primitives.

Versions

Table 1 - Library versions.

Version	Release date	What's new
1.0.0	08.02.2025	- First version.
1.1.0	21.02.2025	- Mutex implementation is improved by putting thread to sleep instead of busy waiting. - Tracer feature is implemented.

Main features

MicroKernel provides a few main concurrency features.

- Main concurrency implementation with a simple scheduler and context switching. Please see the section kernel for detailed information.
- Mutex for critical sections. Please see the the section Mutex for detailed information.
- Semaphore for critical sections. Please see the the section Semaphore for detailed information.

Following diagram shows main code flow of usage :

Kernel interface

Kernel interface is declared in **kernel.h** file.

```
/// Initialize kernel.
int initKernel(int maxNumberOfThreads);

/// Add new thread to kernel scheduler.
int addThread(void (*threadFunc)(), int stackSize);

/// Start scheduler.
int startScheduler(int periodMilliseconds);

/// Runs scheduler to for context switch.
void yieldCurrentThread(void);
```

initKernel method

Initializes the kernel. It is the function that should be called before all kernel related initializations including synchronization primitives creation.

```
int initKernel(int maxNumberOfThreads);
```

Parameter	Value
maxNumberOfThreads	Max number of thead. It is better to not give huge numbers because kernel will allocate thread control blocks as many as max thread number.

Returns: 0 if kernel initialized properly or -1 if there was an error.

addThread method

Adds new thread to kernel scheduler. Before calling **addThread()** please ensure that **initThread()** is executed properly.

```
int addThread(void (*threadFunc)(), int stackSize);
```

Parameter	Value
threadFunc	Function pointer to thread function.
stackSize	Required stack size in bytes for thread.

Returns: 0 if the thread is added properly or -1 if there was an error.

startScheduler method

After initialization of kernel and adding threads, it is ready to start scheduler to start concurrency for these added threads. Please note that **startScheduler** does not return in case of non error situation.

```
int startScheduler(int periodMilliseconds);
```

Parameter	Value
periodMilliseconds	Scheduler interval in milliseconds.

Returns: -1 if it is not started, in case of proper execution this function does not return.

yieldCurrentThread method

Runs scheduler to contex switch. So current thread will be yield and next thread will be run.

```
void yieldCurrentThread(void);
```

Synchronization primitives

MicroKernel provides elementary synchronization primitives mutex and semaphore.

Mutex

Mutex interface is declared in **Mutex.h** file.

```
/// Create an instance.
Mutex_t* Mutex_create();

/// Destroy an instance.
void Mutex_destroy(Mutex_t *self);

/// Lock the mutex.
void Mutex_lock(Mutex_t *self);

/// Unlock the mutex.
void Mutex_unlock(Mutex_t *self);
```

Mutex_create method

Creates the instance of mutex.

```
Mutex_t* Mutex_create();
```

Returns: the address of created mutex or NULL in error.

Mutex_destroy method

Destroys the mutex instance.

```
void Mutex_destroy(Mutex_t *self);
```

Parameter	Value
self	Address of instance.

Mutex_lock

Locks the mutex. In case mutex already locked, it will put current thread to sleep.

```
void Mutex_lock(Mutex_t *self);
```

Parameter	Value
self	Address of instance.

Mutex_unlock

Unlocks the locked mutex. If mutex is already unlock it will return directly.

```
void Mutex_unlock(Mutex_t *self);
```

Parameter	Value
self	Address of instance.

Semaphore

Semaphore interface is declared in **Semaphore.h** file.

```c
/// Create an instance.
Semaphore_t* Semaphore_create(int32_t initialValue, uint32_t maxValue);

/// Destroy an instance.
void Semaphore_destroy(Semaphore_t *self);

/// Release the semaphore.
void Semaphore_release(Semaphore_t *self);

/// Acquire the semaphore.
void Semaphore_acquire(Semaphore_t *self);
```

Semaphore_create method

Creates the semaphore instance.

```c
Semaphore_t* Semaphore_create(int32_t initialValue, uint32_t maxValue);
```

Parameter	Value
initialValue	Initial value of semaphore.
maxValue	Max value of semaphore.

Returns: the address of created semaphore or NULL in error.

Semaphore_destroy method

Destroys the semaphore instance.

```c
void Semaphore_destroy(Semaphore_t *self);
```

Semaphore_release method

Releases the semaphore. It will wake all threads waiting related semaphore. After waking up threads, it will directly force to scheduler choose new thread.

```c
void Semaphore_release(Semaphore_t *self);
```

Parameter	Value
self	Address of instance.

Semaphore_acquire method

Acquires the semaphore. If related semaphore has value 0, this will put current thread to sleep.

Parameter	Value
self	Address of instance.

Tracer feature

MicroKernel supports tracing the scheduler events. It saves events in an internal buffer and calls the user set callback when the buffer is full. So user
can transmit this buffer by any preferred method to host machine and visualize the kernel scheduler events. Please check [MicroKernelTracer]() project, it is an template example that show how to parse this buffer and visualize the scheduler events.

In order to enable tracer feature, please follow the steps :

1. Change **TRACER_OFF** definition to **TRACER_ON** in /include/kernelConfig.h file.
2. Set callback for transmitting data by using :

   ```
   void setSendTracerDataCallback(void (*func)(uint8_t*, int));
   ```

   It is recommended to have a quick transmission in callback function because during this callback all interrupts are disabled so system will no react any coming
   interrupt and scheduler wont run any thread until callback finishes.

Note : If transmission of buffer fails, it might be because of stack size of tracer sending thread, because user set callback is invoked by a internal kernel thread.
So consider increasing TRACER_THREAD_STACK_SIZE from /include/kernelConfig.h file. Also DO NOT change the TRACER_BUFFER_SIZE because current version supports only 1023 events in each buffer.

Simple example

Following code shows simplest example for **MicroKernel** usage. It implements single producer single consumer multi threaded application scenario.

```
#include "kernel.h"
#include "Mutex.h"
#include "Semaphore.h"

#define THREAD_STACK_SIZE_BYTES   100
#define MAX_NUM_THREAD            10
#define SCHEDULER_PERIOD_MS       10

uint32_t g_sharedData = 0;
Mutex_t *g_sharedDataMutex = NULL;
Semaphore_t *g_dataAvailableSem = NULL;

void producerThreadFunc()
{
  while (1)
  {
    Mutex_lock(g_sharedDataMutex);
    g_sharedData++;
    Mutex_unlock(g_sharedDataMutex);

    // Signal that new data is available
    Semaphore_release(g_dataAvailableSem);
  }
```

```
}

void consumerThreadFunc()
{
  uint32_t readValue = 0;
  while (1)
  {
    // Wait until data is available
    Semaphore_acquire(g_dataAvailableSem);

    Mutex_lock(g_sharedDataMutex);
    readValue = g_sharedData;
    Mutex_unlock(g_sharedDataMutex);
  }
}

int main(void)
{
  initKernel(MAX_NUM_THREAD);

  g_sharedDataMutex = Mutex_create();

  g_dataAvailableSem = Semaphore_create(0, 1); // Initial value 0, max value 1.

  addThread(producerThreadFunc, THREAD_STACK_SIZE_BYTES);
  addThread(consumerThreadFunc, THREAD_STACK_SIZE_BYTES);

  startScheduler(SCHEDULER_PERIOD_MS);

  return 0; // Code execution never reaches here
}
```