Task Concept and Heap Management in FreeRTOS

Mojtaba Bagherzadeh, Adrien Lapointe

Royal Military College (RMC)

mojtaba@cs.queensu.ca,adrien.lapointe@rmc.ca

February 11, 2018

Overview

Task Concept

A FreeRTOS application is designed as a set of tasks. A task definition consists of

- Name
- Priority
- Implementation which is defined as a C function, and must not return any value and terminate. It must have a `void *` argument (task function).

```
void ATaskFuntion (void * vpArg)
```

Task Concept

A FreeRTOS application is designed as a set of tasks. A task definition consists of

- Name
- Priority
- Implementation which is defined as a C function, and must not return any value and terminate. It must have a `void *` argument (task function).

```
void ATaskFuntion (void * vpArg)
```

Task Concept

A FreeRTOS application is designed as a set of tasks. A task definition consists of

- Name
- Priority
- Implementation which is defined as a C function, and must not return any value and terminate. It must have a `void *` argument (task function).

```
void ATaskFuntion (void * vpArg)
```

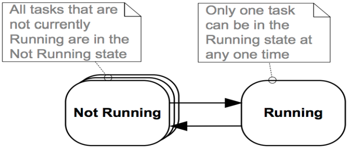
Task Concept

A FreeRTOS application is designed as a set of tasks. A task definition consists of

- Name
- Priority
- Implementation which is defined as a C function, and must not return any value and terminate. It must have a `void *` argument (task function).

```
void ATaskFuntion (void * vpArg)
```

In general a task status can be in `running` or `Not Running` state. In a single core system, only one task can be in running state.



A Typical Task Function

```
void ATaskFunction( void *pvParameters ){
        /* Variables can be declared just as per a normal
            function. */
        int32_t lVariableExample = 0;
        /* A task will normally be implemented as an
            infinite loop. */
        for( ;; ){
                /* The code to implement the task
                    functionality will go here. */
        }
        /* Should the task implementation ever break out of
            the above loop, then the task must be deleted
            before reaching the end of its implementing
            function. The NULL parameter passed to the
            vTaskDelete() API function indicates that the
            task to be deleted is the calling (this) task.
            */
        vTaskDelete( NULL );
}
```

Creating Tasks

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
        const char * const pcName,
        uint16_t usStackDepth, void *pvParameters,
        UBaseType_t uxPriority, TaskHandle_t *pxCreatedTask )
```

Argument	**Description**
pvTaskCode	name of the task function
pcName	task name
usStackDepth	size of stack for the task
pvParameters	a value which is passed into the task function.
uxPriority	priority of the task that can be assigned from 0 (the lowest priority) to `configMAX_PRIORITIES - 1`.
pxCreatedTask	Can be used to pass out a handle to the task being created.

Creating Tasks

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
        const char * const pcName,
        uint16_t usStackDepth,void *pvParameters,
        UBaseType_t uxPriority,TaskHandle_t *pxCreatedTask )
```

Argument	**Description**
pvTaskCode	name of the task function
pcName	task name
usStackDepth	size of stack for the task
pvParameters	a value which is passed into the task function.
uxPriority	priority of the task that can be assigned from 0 (the lowest priority) to `configMAX_PRIORITIES - 1`.
pxCreatedTask	Can be used to pass out a handle to the task being created.

Creating Tasks

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
        const char * const pcName,
        uint16_t usStackDepth,void *pvParameters,
        UBaseType_t uxPriority,TaskHandle_t *pxCreatedTask )
```

Argument	Description
pvTaskCode	name of the task function
pcName	task name
usStackDepth	size of stack for the task
pvParameters	a value which is passed into the task function.
uxPriority	priority of the task that can be assigned from 0 (the lowest priority) to `configMAX_PRIORITIES - 1`.
pxCreatedTask	Can be used to pass out a handle to the task being created.

Return values of the function:

- `pdPASS`: the task has been created successfully.
- `pdFAIL`: the task has **not** been created .

A Typical FreeRTOS Application

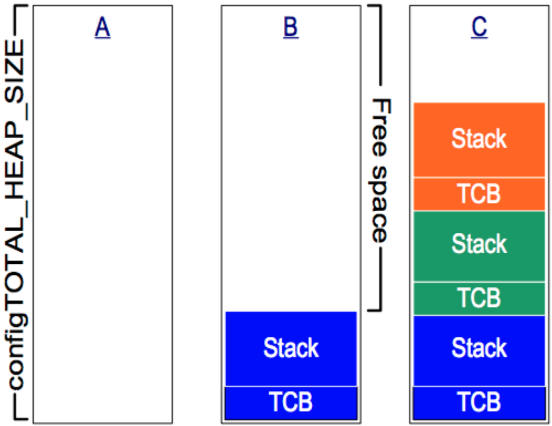
```
int main( void )
{
 // Create tasks. In real scenario you should check the
     return value of the task creation function to make sure
      that task are created successfully
 xTaskCreate(....);
 xTaskCreate(....);

 //  Start the scheduler so the tasks start executing.
 vTaskStartScheduler();

 // In normal situation, the execution should never reach
     here.
 for( ;; );
}
```

FreeRTOSConfig.h

FreeRTOSConfig.h is used to tailor FreeRTOS for use in a specific application. For example,

- `configUSE_PREEMPTION` defines whether the co-operative or pre-emptive scheduling algorithm will be used.
- `configTOTAL_HEAP_SIZE` defines the total heap size of the application.
- `configMAX_PRIORITIES` defines the maximum allowable priority for a task.

Memory allocation for a FreeRTOS application



Data Types

All C basic data types can be used in FreeRTOS application. In addition, two data types are specific to FreeRTOS.

Type Name	Description
`TickType_t`	a data type used to hold the tick count value, and to specify times. It can be either an unsigned 16-bit type or an unsigned 32-bit type, depending on the setting of `configUSE_16_BIT_TICKS`,
`BaseType_t`	the most efficient data type for the architecture. Typically, this is a 32-bit type on a 32-bit architecture, a 16-bit type on a 16-bit architecture, and so on.

FreeRTOS source code explicitly qualifies every use of char with either signed or unsigned, unless the char is used to hold an ASCII character, or a pointer to char is used to point to a string.
Plain int types are never used.

Naming Rules

- **Variables:** Variable names are prefixed with their type: `v` for `void`, `c` for `char`, `s` for `short`, `l` for `long`, and `x` for `portBASE_TYPE` and any other types (structures, task handles, queue handles, etc.).
  unsigned variables and pointers are also prefixed with a `u` and `p` respectively. Therefore, variable of type unsigned char will be prefixed with `uc`, and a variable of type pointer to char will be prefixed with `pc`.
- **Functions:** Functions are prefixed with both the type they return and the file they are defined in. For example: `vTaskPrioritySet()` returns a void and is defined within `task.c`. `xQueueReceive()` returns a variable of type `portBASE_TYPE` and is defined within `queue.c`.
- **Macro:** Macro Names are written in upper case and prefixed with lower case letters that indicate where the macro is defined.

Recall

Stack

The stack is the memory space for a thread (a task in FreeRTOS) of execution. The stack is always reserved in a LIFO (last in first out) order, i.e., the most recently reserved block is always the next block to be freed.

Recall

Stack

The stack is the memory space for a thread (a task in FreeRTOS) of execution. The stack is always reserved in a LIFO (last in first out) order, i.e., the most recently reserved block is always the next block to be freed.

Heap

The heap is a memory block for dynamic allocation. A block can be allocated and freed at any time. This makes it much more complex to keep track of which parts of the heap are allocated or free at any given time; there are many custom heap allocation mechanism for different usage patterns.

Dynamic Memory Allocation in FreeRTOS

Mechanism

FreeRTOS does not use `mallaoc` and `free` for dynamic memory allocation. This is due to the real-time applications requirement such as determinism. Instead it relies on `pvPortMalloc()` and `vPortFree()` which are provided by the portable layer.

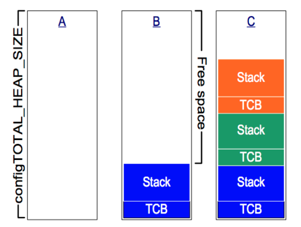
Default Support

FreeRTOS supports five mode of implementation for `pvPortMalloc()` and `vPortFree()` which are called `heap_1` to `heap_5`.

Heap_1

Mechanism

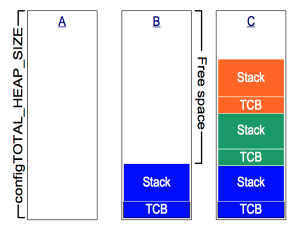
Memory only gets allocated before the scheduler has been started. It subdivides a heap into smaller blocks, as calls to `pvPortMalloc()` are made. The heap size is specified `TOTAL_HEAP_SIZE`.



Heap_1

Mechanism

Memory only gets allocated before the scheduler has been started. It subdivides a heap into smaller blocks, as calls to `pvPortMalloc()` are made. The heap size is specified `TOTAL_HEAP_SIZE`.



Application Area

- Safety critical systems
- Applications that never delete a task

Heap_2

Mechanism

It uses a best fit algorithm to allocate memory and, unlike `heap_1`, it does allow memory to be freed. The best fit algorithm ensures that `pvPortMalloc()` uses the free block of memory that is closest in size to the number of bytes requested.

Heap_2

Mechanism

It uses a best fit algorithm to allocate memory and, unlike `heap_1`, it does allow memory to be freed. The best fit algorithm ensures that `pvPortMalloc()` uses the free block of memory that is closest in size to the number of bytes requested.

Application Area

- Applications that create and delete tasks repeatedly.

Note

- Use `heap_4` instead of `heap_2`.
- similar to `heap_1` the heap is allocated statically when applications start based on `TOTAL_HEAP_SIZE`.
- `Heap_2` is not deterministic.
- It can cause the fragmentation.

Heap_3

Mechanism

`Heap_3` uses the `malloc()` and `free()` functions, so the size of the heap is defined by the linker configuration, and the `TOTAL_HEAP_SIZE` setting has no affect.

Heap_3

Mechanism

`Heap_3` uses the `malloc()` and `free()` functions, so the size of the heap is defined by the linker configuration, and the `TOTAL_HEAP_SIZE` setting has no affect.

Application Area

- Is not recommended for safety critical systems.
- Is not recommended for system with limited resources.

Heap_3

Mechanism

`Heap_3` uses the `malloc()` and `free()` functions, so the size of the heap is defined by the linker configuration, and the `TOTAL_HEAP_SIZE` setting has no affect.

Application Area

- Is not recommended for safety critical systems.
- Is not recommended for system with limited resources.

Note

- Requires the linker to setup a heap, and the compiler library to provide `malloc()` and `free()` implementations.
- Is not deterministic.
- Considerably increase the RTOS kernel code size.

Heap_4

Mechanism

Heap_4 uses a first fit algorithm to allocate memory. Unlike heap_2, heap_4 combines adjacent free blocks of memory into a single larger block, which minimizes the risk of memory fragmentation

Heap_5

Mechanism

The algorithm used by `heap_5` to allocate and free memory is identical to that used by `heap_4`. Unlike `heap_4`, `heap_5` is not limited to allocating memory from a single statically declared array; it can allocate memory from multiple and separated memory spaces.

Heap_5

Mechanism

The algorithm used by `heap_5` to allocate and free memory is identical to that used by `heap_4`. Unlike `heap_4`, `heap_5` is not limited to allocating memory from a single statically declared array; it can allocate memory from multiple and separated memory spaces.

Application Area

- When RAM provided by the system on which FreeRTOS is running does not appear as a single contiguous.

Monitor Heap Free Size

`size_t xPortGetFreeHeapSize( void )`

Returns the number of free bytes (unallocated) in the heap at the time `xPortGetFreeHeapSize()` is called.

`size_t xPortGetMinimumEverFreeHeapSize(void)`

Returns the minimum number of free bytes that have ever existed in the heap since the FreeRTOS application started executing (worst case analysis). It is only supported with `heap_4` or `heap_5` is used.

Handle Heap Failure

If the heap can not be allocated (often because of the size limit), we can have a failed hook function to handle the failure.

- set `configUSE_MALLOC_FAILED_HOOK` to `1` in `FreeRTOSConfig.h`.
- Implement a failure handling function with the following signature:
  `void vApplicationMallocFailedHook( void )`

References

Richard Barry. Mastering the FreeRTOS Real Time Kernel. FreeRTOS.org, 2016

Question?