An Overview of FreeRTOS| With Modifications

Dylan Bertsch

What is FreeRTOS?

The FreeRTOS Kernel is a real time operating system for embedded devices; it is most popular on microcontrollers. The use of an RTOS allows for the tight control of system functions, execution of time critical tasks, as well as the reduction of performance overhead when compared to other embedded solutions, such as Linux. The kernel is composed of several open source components, the most relevant ones are listed in Figure 1.

File Name	Description
Tasks.c and Tasks.h	Contains several data structure definitions, such as the task control block, provides many of the scheduling mechanisms, and many other functions to assist with task management
Port.c or Port.asm	A companion to Tasks.c/h, this is the platform specific implementation for functions such as context switching.
Queue.c/h	Contains definitions relating to the implementation of queues
List.c/h	Contains definitions relating to the implementation of linked lists

Figure 1

The goal of this paper is to examine several FreeRTOS components and demonstrate modifications to the kernel that allow a simple implementation of a system call[1]. It is important to note that FreeRTOS does not natively contain a kernel mode (necessary for system calls); we will examine some possible methods, and their practicality. In order to facilitate development, we use the FreeRTOS simulator[2], built for Visual Studio. This allows us to more quickly debug problems and use various debugging capabilities provided by the WIN32 platform.

FreeRTOS supports many popular processor platforms, and additionally has many derivations for different uses. For example, one derivation(SafeRTOS) was designed to be used specifically in safety-critical applications. These applications can range from aerospace to military defense systems; however, derivations like SafeRTOS usually require more development skills than FreeRTOS due to their safety-critical nature.

In order to achieve the end goal, our journey begins with “detective work”, the following sections will summarize and provide insights of how the FreeRTOS system operates, by looking at their respective source.

[1] A system call is the programmatic way in which a computer program requests a service from the kernel of the operating system. https://en.wikipedia.org/wiki/System_call

[2] Included in the FreeRTOS source, available at https://www.freertos.org/

Literature | The Use of FreeRTOS in Teaching Real-time Embedded Systems

For many, FreeRTOS serves as a professional tool as well as an invaluable learning environment for embedded engineering. **The Use of FreeRTOS in Teaching Real-time Embedded Systems** focuses on the experience of two university professors and their use of FreeRTOS in the classroom. Popular opinion shows that the education community believes FreeRTOS is a good candidate for learning about operating systems and their role in computing. In the above study, FreeRTOS was used to "[emphasize] engineering issues of designing and developing real-time systems in practical embedded applications like automation." (He & Huang, 2014) The authors guided their students with FreeRTOS's rich example projects and its ability to serve as a sturdy development platform to tackle various software challenges. The authors were pleased with their students' learning outcomes and plan to continue their research.

FreeRTOS's well documented nature and mature API gives it an edge when compared to other embedded operating systems. FreeRTOS has been in development for over ten years and consequently has a large developer following and healthy corporate ecosystem. These positive qualities make FreeRTOS a good choice for exposing undergraduate students to industry standard practices; as indicated by the above journal article.

Overview | Task.c

The file Tasks.c contains perhaps some of the most important components needed for FreeRTOS; definitions and functions related to how tasks are scheduled and executed. Before the FreeRTOS system starts, user tasks are added to the task queue by creating their respective task control blocks via the function call "xTaskCreate". The function has inputs for various parameters such as priority, stack size (used for task stack variables), a function pointer (ties a user function to the task), as well as a generic pointer (used for passing input arguments to a function tied to a task, at task startup).

The first procedure in "xTaskCreate"(Depicted in Appendix Figure 3) is to allocate memory space for the TCB's pointers. Such pointers point to a stack, task name, etc. The input arguments are passed into their respective variable locations within the TCB; if everything is allocated successfully "pdPASS"[3] is returned. Later, we will examine how modifications to the TCB structure can help to implement features like syscalls, and other operating system functions.

Tasks.c contains more functions related to scheduler control, as well as general task control. Such functions are called, "vTaskStartScheduler"," SuspendScheduler", etc. After user tasks are added, the user calls "vTaskStartScheduler", which spawns an idle task; from this point on the FreeRTOS kernel controls program execution. The kernel uses priority based scheduling and round robin as a tie breaker. A good chunk of the scheduling logic can be followed from line 2706 in "Tasks.c". The scheduler is complex and out of the scope of this paper; we will consider it a "black box" and accept its functionality without proof.

[3] "pdPASS" – logical flow terms are used throughout the FreeRTOS operating system, they appear as logical types, but are just preprocessor definitions relating to integers. They provide an easier to read codebase.

FreeRTOS contains a few abstractions for lists and queues; they are conveniently located in the files “list.h/c” and “queue.h/c”. Data structures like these prove to be useful for the cohesive design of operating system components, in this section we will look at some implementation examples of lists and queues within the FreeRTOS kernel source.

- Naturally, the scheduler ready queue is a prime example; Figure 2 shows an excerpt of scheduler code; “prvAddTaskToReadyList” is allowing a task to be context switched and have CPU time.

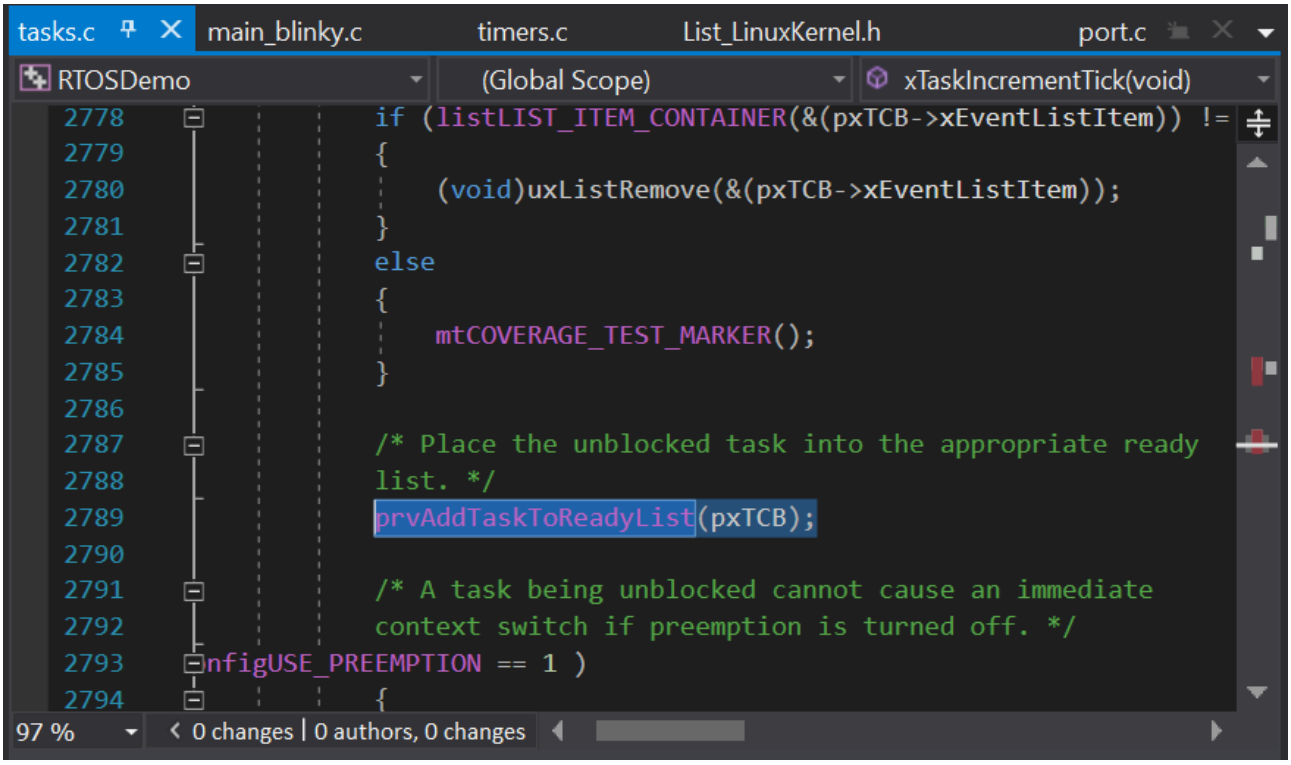
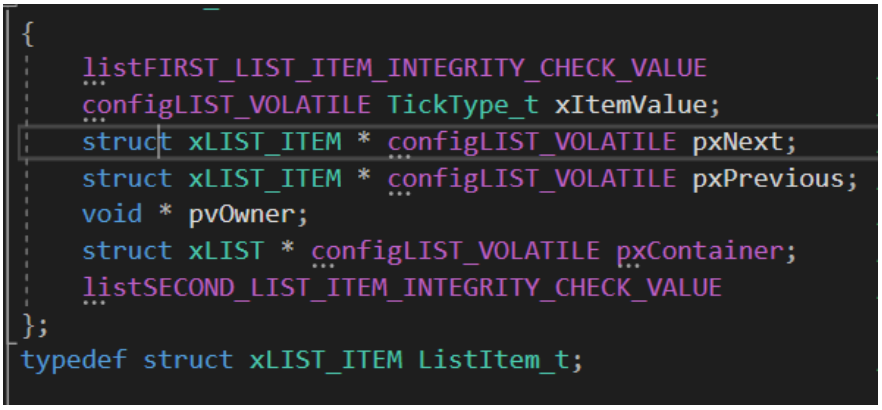


Figure 2

FreeRTOS adopts many standard operating system design practices, and as a result many data structures resemble their desktop counterparts. In Robert Love’s book, “Linux Kernel Development”, linked lists are described as “the simplest and most common data structure in the Linux kernel”. Comparing the Linux implementation to the FreeRTOS linked list data structure; we find them to be very similar. Linux defines its linked list in <linux/list.h> additionally, Linux provides a collection of C macros useful for list manipulation. Later, we will use the Linux list API in our implementation of syscalls.

```
struct list_head {
        struct list_head *next
        struct list_head *prev;
};
```

Figure 2.5 – Linux list definition



```
{
    listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE
    configLIST_VOLATILE TickType_t xItemValue;
    struct xLIST_ITEM * configLIST_VOLATILE pxNext;
    struct xLIST_ITEM * configLIST_VOLATILE pxPrevious;
    void * pvOwner;
    struct xLIST * configLIST_VOLATILE pxContainer;
    listSECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE
};
typedef struct xLIST_ITEM ListItem_t;
```

Figure 3 – FreeRTOS list definition

Queues, like linked lists are an essential building block for operating system design; they facilitate the producer-consumer model, and allow for basic operating system concepts, like inter-process communication. Even though the FreeRTOS kernel is *relatively* simple when compared to desktop operating systems, it incorporates many operating system fundamentals. According to the FreeRTOS documentation, "queues are the primary form of intertask communications. They can be used to send messages between tasks, and between interrupts and tasks".[4] In simple terms, a producer pushes data onto the queue, and the consumer pulls data off the queue. Additionally, regarding memory allocation, both the Linux and FreeRTOS kernels take complete responsibility for allocation the memory used as the queue storage area (look at footnote). The FreeRTOS queue API exposes the functions, "xQueueCreateStatic()", and "xQueueCreate()" in order to facilitate queue memory allocation. The difference between the two functions lies in the difference between dynamic heap allocation vs. compile time allocation. If the queue size is known at compile time, "xQueueCreateStatic" should be used, in most cases "xQueueCreate" can be employed. As a case study, the FreeRTOS queue definition is displayed in Appendix Figure 1.

Overview | Process Synchronization and Inter-task Communication

FreeRTOS, like other modern operating systems includes various mechanisms for process communication and synchronization. These include:

- Mutexes
- Stream and Message Buffers
- RTOS Task Notifications (similar to process signals)
- Binary Semaphores
- Counting Semaphores
- Queues

[4] https://www.freertos.org/Embedded-RTOS-Queues.html

Note: For conciseness, task notifications, Queues, and mutexes are the most relevant.

Real time operating systems need to be fast, reliable, and have *relatively* easy to understand API's. A well implemented IPC structure positively contributes to the above requirements. The FreeRTOS task notification system provides tasks the ability to send and receive a 32-bit *notification value*, this in turn can lead to less RAM usage, as well as up to 45 percent faster performance when using it to unblock a task (direct notification vs. semaphore).

FreeRTOS documentation: Task notifications can update the receiving task's notification value in the following ways:

- Set the receiving task's notification value without overwriting a previous value
- Overwrite the receiving task's notification value
- Set one or more bits in the receiving task's notification value
- Increment the receiving task's notification value

Additionally, if we need to protect a hardware or software resource, FreeRTOS provides mutexes, which are a subclass of a binary semaphore. Mutexes are used to *guard* resources, such resources can be implemented in hardware or software and can include UART (Serial Port), GPIO interfaces, network, etc., they can also include various software resources like shared memory regions, Syscalls, or other system services.

Lastly, queues contribute to the previous inter-process schema, but allow for more object-oriented solutions, like message passing to occur. Queues are essential to good operating system design and are used extensively through the FreeRTOS kernel source. Listed in Appendix Figure 1 is the FreeRTOS queue data structure. Note that the queue implementation includes the use of other mechanisms, like semaphores to maintain synchronization.

Brief Overview | What is a Syscall?

System calls (Syscalls) provide an interface to services made available by the operating system (Silberschatz, Galvin, & Gagne, 2014, p. 62). There are many examples of syscalls, but the most common include functions that deal with memory allocation ("malloc", "free"), and ones that handle I/O ("CreateFile"," ReadFile", etc.). Syscalls are usually privileged operations, which translates to a user program *asking* the kernel if it can do something. Programs need access to various operating system components, and therefore require safe implementations for these operations to ensure error-free execution. Real time operating systems generally do not require the use of syscalls; however, their use can simplify the development of applications and facilitate OS development.

Basic Syscall Implementation | Task Control Block Modifications

In basic terms, our end goal involves having a user thread call a system function and a corresponding kernel thread that the request. For the remaining sections, we will consider a system print function, which takes in a string as an argument, and prints that string to the console.

The first step requires the creation of a kernel thread, however FreeRTOS has no distinction between a user thread and a kernel thread(privileged)[5], so we add an 8-bit unsigned integer called taskUserLevel to the task control block, shown in Figure 5. A value of 0 is assigned to the kernel and a value of 1 is assigned to a user thread. As a result, when a thread is created it is additionally assigned a privilege level, as well as a 50-byte memory space for input arguments("SystemCallParameters[50]"); also shown in Figure 5. The kernel thread (Called SystemCallHandler) is created during the initial scheduler startup process("vTaskStartScheduler"); shown in Figure 6. The kernel thread is attached to a function called "SystemCallHandler", which monitors a linked list for syscall requests.

A syscall request consists of the "SystemCall_struct", defined in Figure 7 and 8, and a list entry("Syscall_ListEntry"). When a user thread makes a syscall, a list entry[6] is created with its corresponding syscallID, and a pointer to its TCB block, the user thread places this list entry on the syscall queue using a sender function(unique to individual syscall); the "printf" example of this is located in Appendix Figure 2.

```
ListItem_t          xStateListItem; /*< The list that the state lis
ListItem_t          xEventListItem;     /*< Used to reference a tas
UBaseType_t         uxPriority;         /*< The priority of the tas
StackType_t         *pxStack;           /*< Points to the start of
char                pcTaskName[configMAX_TASK_NAME_LEN];/*< Descrip
uint8_t taskUserLevel;
void* SystemCallParameters[50];//Keep a table of 50 potential argum
void* SystemCallReturnPointer;
```

Figure 5

```
currentTaskSyscallReturnPointer = NULL;
xReturn = xTaskCreate(SystemCallHandler,
    "SystemCallHandler",
    configMINIMAL_STACK_SIZE,
    (void *)NULL,
    2, /* In effect ( tskIDLE_PRIORITY | portPRIVILEGE_BIT ), but tskI
    &xSystemCallTaskHandle); /*lint !e961 MISRA exception, justified a
if (xSystemCallTaskHandle != NULL)
{
```

Figure 6

[5] Note, while FreeRTOS has a concept of memory protection, it does not allow for a common syscall implementation, and additionally memory protection is not available on all platforms.

[6] We use the Linux implementation for linked lists; Used for compatibility with other operating systems, and additionally to link to the earlier discussion.

```
struct SystemCall_struct
	{
		uint8_t syscallID;//What system call to run
		void* TCB_Pointer;//"who" the task is, we can use this to get the syscall
parameters
	};
```

Figure 7

```
		int systemCallQueueCount = 0;//Keep track of how many items in the queue.
struct Syscall_ListEntry {
	struct SystemCall_struct* syscall;
	struct list_head mylist;
};
LIST_HEAD(SystemCalls_LinkedList);
struct Syscall_ListEntry ListItem;
```

Figure 8

A syscall request needs to satisfy a few requirements:

- Total size of parameters needs to be less than 50 bytes
- Correctly adds request to system call queue
- Increments the system call queue item count
- Wait(block) until the system call is processed by the kernel thread

As mentioned before, the kernel thread is attached to the function “SystemCallHandler”; this thread does the processing part of the syscall process. Continuing the previous example; after the system call is placed on the list, the SystemCallHandler goes through the list and finds system calls to execute. In this case, a user thread is calling the printf system call, the handler then pulls the function arguments from the user thread’s TCB and uses this information to execute some code and stores a return value in the task’s TCB(“SystemCallReturnPointer”). The result is text printed to the console.

Syscall Implementation | Conclusion

In practice, FreeRTOS does not have the need for system calls, RTOSes in general aim to be as streamlined as possible, and as a result desktop operating system features like system calls are dropped. Overall the above syscall implementation is more of an education exercise rather than a practical use case. Even though such a simple syscall approach lacks the pose and performance of a traditional desktop operating system it still serves as a valuable educational exercise in operating system design.

On page 1-2 we discussed FreeRTOS’s place in the teaching community through a few journal articles. FreeRTOS continues to display positive traits for teaching and development in its ability to be modified and examined for educational purposes. Other industry leading real time operating systems

like Micrium and QNX lack the ability to be modified like FreeRTOS; mostly due to the fact that their respective software licenses do not allow modification of their source code. These RTOSes are not able to be downloaded freely, and as a result do not have the same accessibility when compared to FreeRTOS; which can be downloaded freely and has no license restrictions.

Appendix

```
/*
 * Definition of the queue used by the scheduler.
 */
typedef struct QueueDefinition /* The old naming convention is used to prevent breaking
kernel aware debuggers. */
{
	int8_t *pcHead;					/*< Points to the beginning of
the queue storage area. */
	int8_t *pcWriteTo;				/*< Points to the free next place in the
storage area. */

	union
	{
		QueuePointers_t xQueue;		/*< Data required exclusively when this
structure is used as a queue. */
		SemaphoreData_t xSemaphore; /*< Data required exclusively when this
structure is used as a semaphore. */
	} u;

	List_t xTasksWaitingToSend;		/*< List of tasks that are blocked waiting to
post onto this queue.  Stored in priority order. */
	List_t xTasksWaitingToReceive;	/*< List of tasks that are blocked waiting to
read from this queue.  Stored in priority order. */

	volatile UBaseType_t uxMessagesWaiting;/*< The number of items currently in the
queue. */
	UBaseType_t uxLength;			/*< The length of the queue defined as the
number of items it will hold, not the number of bytes. */
	UBaseType_t uxItemSize;			/*< The size of each items that the
queue will hold. */

	volatile int8_t cRxLock;		/*< Stores the number of items received from
the queue (removed from the queue) while the queue was locked.  Set to queueUNLOCKED when
the queue is not locked. */
	volatile int8_t cTxLock;		/*< Stores the number of items transmitted to
the queue (added to the queue) while the queue was locked.  Set to queueUNLOCKED when the
queue is not locked. */

	#if( ( configSUPPORT_STATIC_ALLOCATION == 1 ) && ( configSUPPORT_DYNAMIC_ALLOCATION
== 1 ) )
		uint8_t ucStaticallyAllocated;	/*< Set to pdTRUE if the memory used by
the queue was statically allocated to ensure no attempt is made to free the memory. */
	#endif

	#if ( configUSE_QUEUE_SETS == 1 )
		struct QueueDefinition *pxQueueSetContainer;
	#endif

	#if ( configUSE_TRACE_FACILITY == 1 )
		UBaseType_t uxQueueNumber;
		uint8_t ucQueueType;
	#endif

} xQUEUE;
```

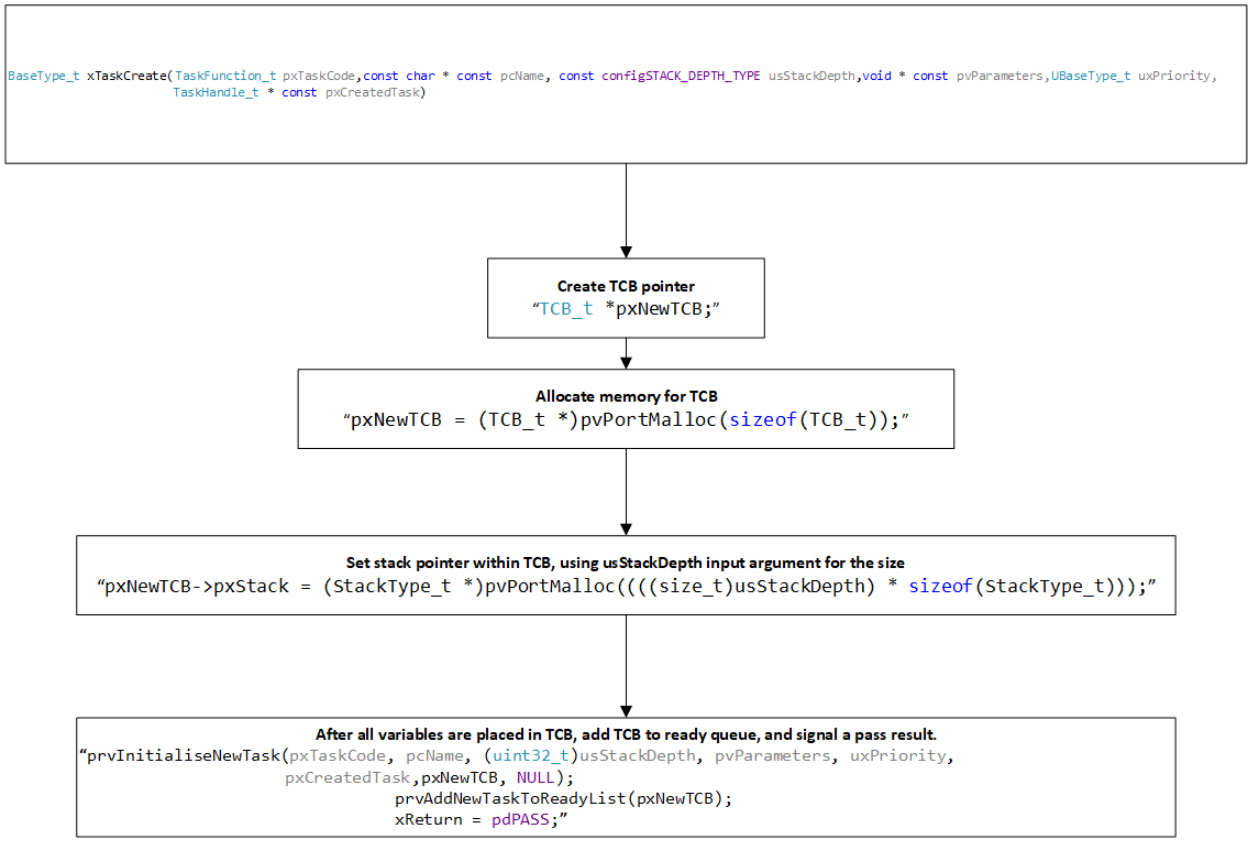
Appendix Figure 1

```
int Syscall_Printf(char* string)//We copy this argument to the TCB syscall argument
block.
{

	struct SystemCall_struct* syscall = (struct
SystemCall_struct*)malloc(sizeof(struct SystemCall_struct));
	syscall->syscallID = SystemCallNumber_Printf;
	syscall->TCB_Pointer = (void*)pxCurrentTCB;
	//Put the parameters from the system call into the TCB
	pxCurrentTCB->SystemCallParameters[0] = string;//since there is only one param,
use slot 1
	//add the system call to the linked list.
	if (systemCallQueueCount == 0)//First Entry
	{
		ListItem.syscall = syscall;
		ListItem.mylist = (struct list_head) { &(ListItem.mylist),
&(ListItem.mylist) };
		list_add(&ListItem.mylist, &SystemCalls_LinkedList);
		systemCallQueueCount++;
	}
	else
	{
		ListItem.syscall = syscall;
		INIT_LIST_HEAD(&ListItem.mylist);
		list_add(&ListItem, &SystemCalls_LinkedList);
		systemCallQueueCount++;
	}
	//Wait for the return value
	int* returnVal = (int*)pxCurrentTCB->SystemCallReturnPointer;
	while (returnVal == NULL)
	{
		returnVal = pxCurrentTCB->SystemCallReturnPointer;
		if(*returnVal == 1 && returnVal != NULL)
		{
			break;
		}
	}

}
```

Appendix Figure 2



Appendix Figure 3

Works Cited

Berry, R. (2009). *Using the FreeRTOS Real Time Kernel | A Practical Guide.*

Cooling, J. (2019). *Real-time Operating Systems Book 2.* Lindentree Associates.

FreeRTOS Documentation. (2019). Retrieved from FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions.

He, N., & Huang, H.-W. (2014, January 1). Use of FreeRTOS in Teaching Real-time Embedded Systems Design Course. *Proceedings of the ASEE Annual Conference & Exposition*.

Love, R. (2010). *Linux Kernel Development* (3rd ed.). New York: Addison-Wesley.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2014). *Operating System Concepts Essentials* (Second ed.). Hoboken: John Wiley & Sons.