REAL-TIME HIBERNUS: AN RTOS WITH COMPUTE THROUGH POWER LOSS FUNCTIONALITY

September 2017

MSc Embedded Systems
ABSTRACT

Energy harvesting is fast becoming a viable source for powering low power devices. A major challenge to this technology is dealing with intermittent nature of energy harvester outputs. One method is to overcome this obstacle is to use super-capacitors or batteries to smooth out the harvester output; another method which is also the focus of this paper is Transient-Powered Computing (TPC). One of the algorithms which deals with Transient operation is known as Hibernus. This approach ensures an application will run during intermittent energy harvester supply. This algorithm stores system state once when energy is low and puts the system to sleep until the supply is restored to an acceptable threshold. The algorithm operates on a processor with FRAM nonvolatile memory. This report details on how to use Hibernus implemented as an Real-Time Operating System (RTOS).RTOSes are most effective in embedded systems used in Automotive industries, medical systems, fire safety systems etc because they ensure meeting task completion deadlines. The idea here is to improve the efficiency of the Hibernus software running in an intermittent supply environment the same way RTOSes are used to improve the performance of very time critical real time embedded systems.
# CONTENTS:

1. **INTRODUCTION** ......................................................................................................................... 1  
   1.1. Project specifications ............................................................................................................. 3  
2. **LITERATURE REVIEW** ............................................................................................................. 4  
   2.1. Hibernus ............................................................................................................................... 4  
   2.2. Ferro-electric Random Access Memory (FRAM) ............................................................... 5  
   2.3. Real-Time Operating System (RTOS) .................................................................................. 6  
      2.3.1. Context Switching ......................................................................................................... 8  
      2.3.2. Task Synchronization .................................................................................................. 9  
      2.3.3. FreeRTOS .................................................................................................................... 9  
   2.4. MSP430 microcontrollers .................................................................................................... 10  
      2.4.1. Low Power Modes (LPM) .......................................................................................... 10  
3. **IMPLEMENTATION** .................................................................................................................... 12  
   3.1. Proposed flowchart of Hibernus-RTOS system .................................................................. 12  
   3.2. RTOS size constraints ......................................................................................................... 13  
   3.3. MSP430 FreeRTOS Port Configuration & MSP430 system configuration ..................... 15  
      3.3.1. System tick Generator .................................................................................................. 16  
      3.3.2. Dynamic memory allocation ....................................................................................... 17  
      3.3.3. Low power modes and Idle task .................................................................................. 17  
   3.4. Functional block diagram .................................................................................................... 18  
      3.4.1. Context Saved even in power loss ............................................................................. 21  
      3.4.2. Energy-Aware Scheduler ........................................................................................... 21  
4. **PERFORMANCE AND ANALYSIS** ......................................................................................... 22  
   4.1. Experimental Setup .............................................................................................................. 22  
   4.2. Analysis of the Startup of the RTOS ................................................................................... 23  
   4.3. Store and Restore operations .............................................................................................. 25  
   4.4. Energy Trace++ ................................................................................................................... 27  
5. **PROJECT MANAGEMENT** ....................................................................................................... 33  
   5.1. Time Analysis ...................................................................................................................... 33  
   5.2. Cost Analysis ...................................................................................................................... 34  
6. **CONCLUSION** ......................................................................................................................... 35  
**REFERENCES** ............................................................................................................................. 36
CHAPTER ONE : INTRODUCTION

The world of smart sensors is becoming more predominant today, in the future the number of sensors is set to reach trillions worldwide [1]. These embedded systems will be operating in the µW power range, the size of the entire device is also a critical parameter which means that if these devices are battery powered, the battery maintenance becomes a problem in a sensor heavy environment. In other words, Battery technology is not advancing at the same pace as the technology it is powering, even in the best case scenarios they have a significant chance of failing, they are not reliable. A networking paradigm of such an environment is called Internet-of-Things (IoT). It enables a diverse range of applications by allowing these smart embedded devices to connect through the internet[2]. IoT’s themselves will operate tens of thousands of autonomous devices(Networked Embedded devices), organized as smaller networks[3]. Rather than connecting these complex distributed networks to a central power grid, it is more sensible for them to be operated by energy harvesters[4]. The general block diagram of such a system is given in Fig1.1.

![Energy Harvesting System Diagram](Image)

**Fig 1.1 An Energy harvesting system with Battery Management (Source : “Overview for Energy Harvesting & Solar Charging”, http://www.ti.com/lsds/ti/power-management/energy-harvesting-and-solar-charging-overview.page)**

Here another problem arises; the battery management +battery circuitry will become a hindrance if the system comprises of multiple MCUs and if each of them have hundreds of sensors because a network of rails and interconnects to multiple batteries have to be made. Moreover, the scaling of energy densities in battery design has not come even close to the
scaling of CMOS device geometry, which is further hindrance to the growth of IoT devices. An alternative method would be monitoring the harvester output directly by the MCU and storing the system state in Non-volatile memory if the output below a threshold, the MCU can restore the state once power is back to an acceptable range without battery management, this is the principle of Transient Powered Computers (TPC)[5]. The focus of this report is dealing with one of the TPC algorithms, Hibernus[6]. Some of the other algorithms are Mementos and Quickrecall[7] -[8]. Mementos transforms general-purpose programs into interruptible computations that are protected from frequent power losses by automatic, energy-aware state check-pointing, this method requires considerable overhead due to high erase/write times of flash memory, Quickrecall improves upon this by making the check-pointing system very lightweight and introducing FRAM, an emerging non-volatile memory technology that combines the benefits of SRAM and flash, to seamlessly enable long-running computations in TPCs. Hibernus uses FRAM but only has one checkpoint that is determined by user-set thresholds. The software for these algorithms are implemented in C programming language. Because of the Asymmetric structure of the algorithm and a degree of randomness involved in the transient behavior of the energy harvester output, it is logical in theory to incorporate a Real-time Operating System into Hibernus. To elaborate, An RTOS has all of the necessary tasks of the Hibernus software in a ready or waiting state (with an interrupt acting as the trigger) whereas ordinary C-code needs to create stack frames for the functions of Hibernus every time an interrupt occurs. An improved version of Hibernus is also available and is called Hibernus++, it introduces dynamic threshold to the Hibernus system i.e, it becomes self-calibrating[9].

Chapter Two explains the theory behind Hibernus including the required hardware, input parameters, overall performance, and the structure of the software (C code) and the comparison of its operation versus other Transient algorithms. This chapter also explores a lot of the core RTOS concepts which are essential to the project and some essential theory pertaining to the hardware.

Chapter three will look into the implementation of the project, structure of the code, the application it is running, which is chosen to be Advanced Encryption Standard (AES). AES is becoming important in IoT because threats to cyber security of embedded devices is leading people to staying away from producing IoT devices, AES is also a resource intensive operation, this gives an opportunity to analyze such an operation running Hibernus. Here we will also look into the hardware used, namely the TI development board, the freeRTOS port for the development board and changes to Hibernus that had to be made in order to carry out the proposed research experiment.
The Testing and analysis of the practical work will be detailed in Chapter four, here the tests for intermittent Transient computing will be performed using a test setup and then detail the analysis of the tests and the inferences gained from the tests.

The final chapter titled “project management” will show the proposed Plan of the project as a PERT chart, with time allocated for the specifications of the project and the actual final plan. Here the reasons for deviations from the original plan will be explained and also give some brief plans about improving this project based on what was learned through research.

1.1 Project Specifications

The main objective of the project is to see whether the Hibernus system works as an RTOS or not and to see whether an RTOS is viable in an intermittent energy harvester power supply environment.

Here are the specifications for this project:

- Do background researches into TPC algorithms and RTOSes like the Key RTOS objects used to improve system performance compared to a non-RTOS software.
- The hardware development platform used is the Texas Instruments MSP430frxx-series experimenter board and the RTOS used is freeRTOS, so making a port of the RTOS for the hardware is the first step in the implementation stage.
- The Hibernus code itself has to be ported since the hardware platform is different from the one used in the original research.
- The Hibernus code is used as a reference, meaning although the algorithm is same, a lot of changes have been made to the base C-code mainly due to change in hardware (Hibernus uses a different dev board). This also gives an opportunity to possibly improve efficiency of the original Hibernus C-code.
- Start adapting the Hibernus algorithm into the RTOS port but without any tasks and only functions, this is to make sure that Hibernus is ported into the new hardware.
- Implement the operations of Hibernus as tasks and check if the system works on a non-transient power loss environment.
- Analyze the performance of the system in a transient environment similar to the one in the Hibernus research paper.
- Identify areas of the code that can be improved for transient operation. This will be explained in the analysis chapter.
CHAPTER TWO: LITERATURE REVIEW

2.1 Hibernus

Hibernus stores the state of the system when power is below a threshold, so the whole algorithm itself has two states namely active and hibernating[6]. During active state the system should operate like any other normal embedded device but at the same time monitor for voltage drops through a peripheral like a comparator or ADC channel. Other approaches like mementos[7] follow a static check-pointing system which inserts specific points in the code(at start of function calls, before loops etc) to save a snapshot. Hibernus is vastly more efficient than this approach as the checkpoints are only taken at the most appropriate time. Fig 2.1 illustrates the Hibernus approach. Hibernus which is the primary focus of my research requires a Non Volatile Memory(NVM) better than flash and the NVM used here is called Ferro-electric Random Access Memory(FRAM).See section 2.2.

![Flowchart explaining the Hibernus algorithm](image)

With this information we can identify some common characteristics to transient computing algorithms. They are as follows:

- **Snapshot**: Copying the context of the micro-controller into FRAM, when a checkpoint is activated.
- **Checkpoint**: During program execution, if the voltage supply is dropping the system makes a decision to take a snapshot.
- Hibernate: Take snapshot and go into low-power mode. Note: snapshots are not taken only in this instance, they are also taken for cases were the power will turn off the device.
- Restore: After power is restored, the system context which is saved in FRAM is restored and the system will continue operation, from where it was left off before Hibernate.

To elaborate more on what “system context” means, context usually encompasses CPU registers (general purpose registers + PC) and a local stack memory (from the programmers view), the local stack in embedded devices translates to the RAM (and local caches in some cases).

Since the hibernate action performed by Hibernate can be performed during any time during the execution of an application, it is considered to be application independent and any software overhead is abstracted away to the programmer.

### 2.2 Ferro-electric Random Access Memory (FRAM)

FRAM also known as FeRAM or F-RAM, is a memory technology that combines the best of flash and SRAM technologies[10]. It is non-volatile like Flash, but offers fast and low power writes, the write endurance is of the order of $10^{15}$ cycles. It also offers bonus features that are very helpful to embedded engineers such as better data security (than flash/EEPROM), resistance to radiation and electromagnetic fields and better code flexibility as a result of FRAM’s unified memory map structure. The following table given by Fig 2.2 shows how FRAM stacks against more traditional NVM technologies (Based on devices by Texas Instruments).

The table shows that Flash is ill suited to dealing with the intermittent nature of Energy harvester outputs, because it cannot deal with the frequency of Store-Restore operations of TPCs (Write speed of 1sec is too slow). The Texas Instrument’s MSP430 is the only processor architecture which has a development board that comes with FRAM memory (write speeds of <10ms). Hence the development board used is the TI EXP-MSP430fr6989 Launchpad development board.

2.3 Real-Time Operating System (RTOS)

A Real-Time Operating System (RTOS) is an Operating System (OS) specifically designed for embedded environments where Real time applications (usually termed tasks) are dealt with regularly. Tasks are actions (written as programs) needed to be performed by the RTOS within a deadline. If the task will not complete within this deadline and finishes after it, it may as well be considered as failed in most cases (depends on the type of application) therefore this leads to the following classification of RTOSes:

- Hard RTOS: here all the tasks must complete within the set deadline or they are considered to have failed i.e., deadline misses cause system failure. Ex: Medical systems, fire safety systems etc.
- Soft RTOS: Here the tasks can complete after the deadline. The RTOS as a whole is considered not to have incurred a system failure due to tasks missing their deadlines. Ex: Setup boxes, Remote control etc.
- Firm RTOS: Some tasks may lead to system failure.
Hibernus is an example for Firm RTOS or Hard RTOS, the application with Hibernus running alongside it can be Hard or Soft whereas Hibernus itself is a Hard task, but the deadlines depend on energy thresholds, this type of RTOS is known as energy-aware RTOS[16]. From all of this we can see that another property of an RTOS is scheduling tasks, scheduling is handled by a set of programs or services called the Kernel, built into the RTOS and is mostly abstracted away from the programmer’s point of view. In an RTOS, only one task can run at a time and it is said to be in the running state. In the course of execution tasks can be in three states namely,

- Running: the task is currently being executed
- Blocked: the task tries to go into running but another task is currently in use, so is blocked
- Suspended: a task is intentionally (by the programmer) taken out of running state and put into a state of stasis and is only taken out of this by the user and not the RTOS kernel.
- Idle : This is a special case where none of the user defined tasks is running or need to be woken up, so the RTOS employs an “Idle task” that is created when the RTOS scheduler is started to ensure there is always at least one task that is able to run. In a pre-emptive scheduler, idle task has the lowest priority.

The kernel facilitates moving the tasks through these states. The programmer can decide the rules which the kernel will follow ie, the scheduling algorithm. Scheduling algorithms can broadly be classified into:

- Co-operative scheduling: Here all tasks share an equal time slice, which is selected by the programmer. This algorithm is considered to be the most “fair” since all tasks get equal amounts of available resources. Ex: Round-robin algorithm
- Pre-emptive scheduling: The tasks in this system are assigned priorities and are scheduled according to priorities; highest priority task will always run at any given time unless the program puts it into a blocked state. There is a problem that will arise here where only the highest priority task will run and devourer all the resources, so the programmer needs to perform task synchronizations in order to ensure that the higher priority tasks will run only when required and will move into “blocked” state awaiting the next call to be run. This algorithm sacrifices “fairness” for ensuring the RTOS adheres to being a Hard RTOS. Ex: Rate Monotonic Algorithm(RMA).

The Hibernus-RTOS has to have a Pre-emptive scheduler, because the hibernate, restore tasks are high priority tasks which should run its entirety without any interruption. Pre-emptive scheduling is best explained by [11].
2.3.1 Context Switching

Every Task has its own stack frame, its own usage of processor registers ie, it utilizes the processor/microcontroller registers and accesses RAM and ROM just as any other program. These resources together comprise the task execution context. The execution of tasks are in a sequential manner and as explained before it does not know when it is going to get suspended or blocked by the kernel(swapped out) or resumed(swapped in) by the kernel, it is also worth noting that it doesn’t even know when this has happened. Consider the example shown in Fig2.3 where the task is suspended immediately before executing an instruction that sums the values contained within two processor registers. While the task is suspended other tasks will perform their own operations with another context and may use the same processor registers and modify register values, so when the suspended task resumes operation, it is unaware that the processor register values have been altered and if it used these values the ADD operation will give wrong results. To prevent this case, the kernel stores the processor register values every time a task gets suspended and restores it when it resumes operation. This is called context switching.

Fig 2.3 An example of RTOS context switch when the task is about to perform ADD
(Source: “Context Switching” http://www.freertos.org/implementation/a00006.html )
2.3.2 Task Synchronization

One of the main advantages of using an RTOS is its ability to ensure only one task will run at a time and that it will not corrupt data of other tasks, this is called Task Synchronization or Inter-task communication, the latter indicates a form of “handshaking” between tasks so that they make it known to the kernel which ones should be blocked or run. The objects commonly used for inter-task synchronization are:

- RTOS Task Notifications
- Queues
- Binary Semaphores
- Counting Semaphores
- Mutexes, and
- Recursive Mutexes

Task Notifications in freeRTOS can be used as a “lightweight” version of other objects. Lightweight in the sense it is faster and consumes less RAM. For more information on task synchronization refer [12].

2.3.3 FreeRTOS

Presently FreeRTOS is the most widely used commercial RTOS available in the market primarily because it is as the name suggests free. All the code comes with an open source GPL license (GNU General Public License).

Other commonly used RTOSes like Vxworks is developed as proprietary software, meaning its entire source code is not public and it is too expensive for a research project such as this. The Texas Instruments Integrated Development Environment (IDE) provides its own RTOS know as TI-RTOS but lacks the documentation and support that freeRTOS provides. The core freeRTOS files are just 3 .c files and the dynamic memory allocation is a separate .c file depending on which implementation of malloc[13] (the dynamic memory allocation function used in C-lang) the programmer chooses. To clarify the previous point, freeRTOS provides its own malloc implementation because the inbuilt C-lang malloc is a library function that is not designed for embedded applications and takes a lot of space (relative to embedded application sizes) and has a comparatively high failure chance, consequently this makes malloc not thread safe, meaning malloc is highly inefficient and unsafe when used in multi-thread\multi-task environments.
2.4 MSP430 microcontrollers

RTOSes are mostly implemented in 32 bit ARM cortex micro-controller architectures, primarily due to the abundance of RAM and Non-Volatile Memory(NVM) but all the architectures come with flash as the NVM. This rules out ARM dev boards for use in Transient Powered Computing(TPC) applications at the moment. MSP430 micro-controllers are 16-bit RISC machines.

Texas Instruments(TI) have a series of low power microcontroller architectures specifically made for energy harvesting applications they are known as MSP430frxx series. These boards come with 16-bit CPUs and come with FRAM NVMs, the downside is less RAM and less ROM(on some CPUs) memory which means FreeRTOS can run only on some of their microcontroller architectures.

The MSP430 used in this project is msp430fr6989 which comes with 2Kbytes of RAM and 128Kbytes of FRAM. The specific reason for choosing this board is explained in Chapter three (section 3.2). The board used in the original Hibernus project is MSP430fr5739 which has 1Kbytes of RAM and 6Kbytes of FRAM.

2.4.1 Low Power Modes(LPM)

Low power modes in MSP430 microcontrollers are designed to give very low leakage current and it operates from a single Vcc supply. This feature gives an extremely low current drain when the system enters into the LPM. There are different LPMs according to the features that each mode disables. As the LPM number rises, the number of things disabled on the chip also rise:

- LPM0 - The CPU is disabled.
- LPM1 - The loop control for the fast clock, Master clock(MCLK) sourced from high frequency digital oscillator(DCO) is also disabled. Highest frequency of MCLK = 16MHZ.
- LPM2 - The fast clock(MCLK) is also disabled.
- LPM3 - The DCO oscillator and its DC generator are also disabled.
- LPM4 - The crystal oscillator is also disabled. The Crystal oscillator sources the ACLK of the device which has a maximum frequency of 50KHz.

This is a general overview of LPM modes, for detailed explanation refer MSP430fr6989 datasheet[14].

The circuitry for the brownout reset is active during all low Power Modes as well supply voltage supervisor circuitry (SVS). Most peripherals can be activated in LPM3 that uses ACLK and peripherals which do not use a clock can be operated in LPM4 for ex: Comparator. This
makes the comparator as the preferable peripheral to use for monitoring the Hibernus
thresholds.

LPM3.5 AND LPM4.5 are “deep sleep” modes where even the SVS is switched off and requires a
use-programmed system reset signal to wakeup. As the LPM mode rises power consumption
decreases, but the time needed to wake up increases although the MSP430 is designed to keep
the worst case wakeup time fairly low. For ex: the clocks which use Frequency Locked Loop(FLL)
system clock module need only a few microseconds to get “locked” once more.
CHAPTER THREE: IMPLEMENTATION

3.1 Proposed flowchart of Hibernus-RTOS system

Fig 3.1 Hibernus RTOS flowchart
The chapter follows the steps for the implementation of the proposed RTOS system chronologically.

Fig3.1 shows the program flowchart of the proposed Hibernus-RTOS design. This is what the system should look like. The Hibernus operation here would save contexts of the tasks(irrespective of what state they are in) into FRAM and restore them after supply returns. The big change here is the fact that Restore doesn’t restore the state of the kernel, meaning the kernel has to be restarted every time. In Hibernus there is no piece of code being restarted, the execution starts from where it stopped before hibernating. Ideally we could save the kernel to maximize the performance of the system but it would require massive changes to the freeRTOS kernel code and not for a huge performance gain because the kernel takes approximately 3ms to start, in comparison the boot-loader of the MSP430 takes 2ms. An alternate would be building an RTOS from scratch which is specially suited for TPCs.

3.2 RTOS size constraints

The MSP430(fr5739) processor used for the Hibernus project has just enough memory for a bare-metal implementation of freeRTOS, 16Kbytes of Non-Volatile memory and 1Kbytes of RAM. For my research purposes the freeRTOS implementation needed atleast 14Kbytes of Non-volatile memory(excluding the space required for the tasks and user-defined global variables). For these reasons I chose a board with 2Kbytes of RAM and 128Kbytes of Non-volatile memory, msp430fr6989.

The default linker file for the board does not come with one contiguous 128KB block of memory, so it is difficult to assign user-defined variables to the FRAM space, since it could cause Memory boundary violations, MPU violations etc, so I modified the default file by moving around some of the default boundaries as wells allocating more space for code segments, since freeRTOS takes around 10KB of memory (without dynamic memory allocation).

Fig 3.2 shows a modified code snippet of the linker file, it shows the memory organization of the MSP430fr6989. The original(default) linker file did not have a single memory block of FRAM and it was divided into two blocks having boundaries with memory which was not FRAM, this was a major obstacle in running the Code, not only was the access all of the FRAM space possible without hassle, freeRTOS was not able to store its .text, .data, .code segments due to inadequate memory. The FRAM+DATA_FRAM segments are used to store the global variables and functions used in the Code, it is also used by the FreeRTOS core modules to store some of its functions and variables ie, this is the new space created for the .text, .code and .data segments.
The FRAM2 segment is used for all user-defined variables key to Hibernus operation. The RAM segment which is 2048 bytes wide will be moved into FRAM2 segment every time a “store” operation occurs due to Hibernus’s operation. The same is true for memory locations belonging to the PERIPHERALS_8BIT and PERIPHERALS_16BIT segments since they contain the Special Function Register(SFR) memory locations. This translates to the following macro’s defined in the main.c file.

```c
#define FRAM2_START 0x20000
#define FRAM2_END 0x24000
#define RAM_START 0x1C00
#define RAM_END 0x2400
#define PC 0x20000
#define RAM_LOC 0x1C00
#define RAM_BACKUP_LOC 0x20000
#define RAM_SIZE 2048
```
The TINYRAM is an experimental feature present in the board to explore the possibility of running a deep sleep mode with a very small memory element acting as the RAM. The main RAM will be powered down during this deep sleep.

Fig 3.3 shows the layout of this project’s program memory with the text, data and stack+heap, the left shows the RAM layout and the right shows FRAM. The text segment (code segment) of the program is stored inside the FRAM2+DATA_FRAM segment, this is not unusual for embedded systems.

### Fig 3.3 Program memory layout (left side: RAM, right side: FRAM)

#### 3.3 MSP430 FreeRtos Port Configuration & MSP430 system configuration

FreeRTOS has a few C-files which contain the source code for the RTOS in question, these files need to be ported for the specific MSP430 architecture, most of the major settings can be edited in a convenient header file called FreeRTOSConfig.h but I also had to make some additional changes namely for system tick generation and Dynamic memory allocation.
3.3.1 System tick Generator

The time keeping in the RTOS is not in seconds since a second is too slow a unit for time measurement in most real time environments. In RTOS the unit for time measurement is called a tick and it is user definable. The tick for MSP430 is generated using one of the timer modules available and uses the 32KHz crystal oscillator, present on the MSP430 development board for clocking the timer. The number of ticks defines the time slice that the tasks get to use for their operation. At the end of the tick a tick interrupt is generated, the objective of this interrupt is to supervise the kernel operation ie, to check the states of the tasks that are in the RTOS(states being: Running, ready, blocked and suspended) and allow the kernel to take appropriate action on the tasks. For example, this project uses a CRC task which runs every 10 ticks, so it has to be blocked for 10 ticks and runs on the 11th tick. The kernel performs this using the tick interrupt mechanism. Fig 3.4 shows the operation of tick interrupt in a freeRTOS demo code.

Fig 3.4 An example of tick interrupt(Source : “The RTOS tick”, http://www.freertos.org/implementation/a00011.html)

- At (1) the RTOS idle task is executing, meaning all higher priority tasks are blocked.
- At (2) the freeRTOS tick occurs, and control transfers to the tick interrupt ISR (3).
- vControlTask and vKeyHandlerTask are generic freeRTOS tasks used here to illustrate tick operation. The RTOS tick ISR makes vControlTask ready to run, and as vControlTask has a higher priority than the RTOS idle task, switches the context to that of vControlTask.
- As the execution context is now that of vControlTask, exiting the ISR (4) returns control to vControlTask, which starts executing (5).

**Tick ISR Pseudo Code:**

```c
TickISR()
{
    Increment tick count
    If( Tick increment readied task)
    {
        Switch execution context to readied task.
    }
    Return from ISR
}
```
A higher tick frequency (consequently Idle tasks run more frequently) means time can be measured to a higher resolution. However, a high tick frequency also means that the RTOS kernel will use more CPU cycles to be less efficient. The RTOS demo applications that is supplied by freeRTOS all use a tick rate of 1000Hz. This is more than enough for most applications running at a CPU speed of 8MHz, the tick rate in this project is chosen to be 500Hz, this setting is part of “FreeRTOSConfig.h”.

3.3.2 Dynamic memory allocation

The Hibernus code does not test the effects of using the in built C-language library for dynamic memory allocation. Although embedded applications do not frequently use dynamic allocation, it is needed in some areas. The freeRTOS malloc function can be ported to non-RTOS code and is a good substitute to use in Hibernus.

As discussed earlier in section 2.3.2, freeRTOS does not use the pre-defined malloc function; it instead has its own malloc which is used in task creation. This malloc creates dynamic memory from a single large block of memory called ucHeap (a single array). For this MSP430, since there is an abundance of FRAM memory, the ucHeap block is allocated to FRAM more specifically the FRAM2 block in the linker file. The following code snippet shows the declaration of ucHeap.

```c
#ifdef __ICC430__
    __persistent
#else
    #pragma NOINIT( ucHeap )
#endif
uint8_t ucHeap[configTOTAL_HEAP_SIZE];
```

In the project ucHeap is chosen to be 7*1024 bytes. This is non-initialized memory so that it doesn’t add to the C-startup procedure time.

3.3.3 Low power modes and Idle task

The IDLE task is programmed to put the system in LPM4, but the system only goes into LMP3 because the tick generator is run by ACLK (refer sections 2.4.1 and 3.3.1). The significance of this is that the RTOS when not in active mode (User defined tasks are not doing anything) is put into low power mode until the hibernate interrupt wakes up the system.
3.4 Functional block diagram

The following section describes the practical RTOS that is used for experimentation. The RTOS has five tasks they are arranged according to their task priority as follows:

- Restore
- Hibernate
- AES(Encrypt)
- AES(Decrypt)
- CRC

In the default case which is after a system reset, the user software must initialize the device, the device has an option to initialize the hardware registers before the C-startup (and Global Variable initialization), in software this is implemented as a function called _system_pre_init(void). Additionally, if the Restore operation takes place inside this function, then Restore will take place before the C-Startup. An additional delay for “kernel initialization” also has to be considered, practically this delay cannot be entirely avoided as is explained later in section 3.4.

The board requires at least 1.8Volts at Vcc at approximately 8mA (Considering the Usage of onboard LEDs as well as the LCD display) to be able to run any application. When the Vcc drops below this threshold the board switches off, at the hardware level this generates a signal for Brown Out Reset (BOR). The hibernate threshold is user defined and is set to activate a store before the Vcc drops to 1.8V. The threshold is programmed to produce a system interrupt that will activate Hibernate task that will save the system state. The aforementioned operation will take place irrespective of any application the system runs. In this Project, the applications allocated as tasks are AES and CRC. The Hardware has a built in Real Time Clock (RTC) and a Liquid Crystal Display (LCD). The RTOS displays the RTC time on the LCD display, just as a physical indicator to affirm Hibernate operation is taking place, because if Hibernate is successful it will store the RTC time when a store operation takes place, since RTC is part of the special function registers. Note: none of the LCD and RTC related functions are implemented as tasks, they are normal C-fuctions.

In Hibernate, “Hibernate” is implemented as a normal C function and is called from the comparator module (COMP_D of msp430fr5739). In the RTOS, Hibernate is a task which once created goes straight in to Blocked state. Once the input to the system crosses a user defined fixed threshold, the internal comparator (COMP_E of msp430fr6989) triggers an interrupt which unblocks the Hibernate task and is put in the Running state. This is in contrast to Normal Hibernate operation where an interrupt will call the Hibernate function, which will create a stack frame for the function call. As I stated in Chapter 2, tasks are created at the start of the
program, no stack frames are created when a task is moved into running state from a block state, only a context switch between tasks. The accuracy of the internal comparator is not great, it has only three pre-set references and threshold changes might cause spurious interrupts, but it is sufficient to test a prototype.

The use of “Task Notify” and “Queues” are for inter-task synchronization purposes (as mentioned in section 2.3.2) and I’m using these objects inside the Interrupt handlers of the various peripherals in use like the comparator. This is in contrast to Hibernus where the hibernate function is called with the Interrupt Service Routines (ISR). ISRs are special functions that do not return any value and can be interrupted by other higher priority ISRs meaning whatever code written in ISRs may not be executed, this is why it is good practice to keep operations in ISRs as small as possible. The Restore need not be implemented as an RTOS task because the system state can be restored before the kernel has even started. While this operation might ensure a working system, it is not the best way to restore the system state in this project for the following reasons:

1. The task related variables (task parameters) are not stored in FRAM; it is loaded on to the RAM, while the tasks themselves can use variables which are stored in FRAM. So ideally all the task parameters need to be stored in FRAM during hibernate task.
2. Along with task parameters, all the variables of the kernel itself should be stored during hibernate. This would mean that the kernel’s state is stored, but starting the kernel during every restore does not create a large overhead.

As it stands neither of these operations are performed, so when the system state is restored, all the task contexts are restored but the kernel is not properly restored and needs to be “restarted”.
The Hibernate task when unblocked, stores the core CPU registers including the Program Counter (PC), all the core Special Function Registers (SFRs) of all the peripherals and the RAM. The board has 16 comparator channels, meaning up to 16 different energy harvesters can be monitored. There are other ways to monitor the harvester output like ADCs. The functional block diagram is given in Fig 3.3.

The CRC task is used to check that during the operation of the scheduler, no timing violation occurs. It is configured to run every 10 ticks of the scheduler. The CRC task uses in-built hardware CRC module to store a calculated CRC value in a variable (very low performance overhead) and the same calculation is performed using software and stored in another variable, the two are compared. If the CRC check fails that means the critical sections in some tasks are being violated and data is getting corrupted.
From a performance point of view, the AES task performs software implementation of the AES encryption. It is divided into two tasks one for encryption and the other, decryption. The decryption takes 9000 CPU cycles and encryption is approximately 7000 cycles [17](with a key length of 15). So when hibernate is activated, the task state is stored and restored when power comes back on.

The Advanced Encryption Standard (AES) is a symmetric block cipher that is the current unbreakable encryption standard that replaced the previous standard known as Data Encryption Standard (DES). Symmetric ciphers use the same key for encryption and decryption meaning the sender and receiver must both know and use the same secret key. One of the key challenges for embedded devices which are connected to internet (IoTs) is data security and implementing a secure AES in these devices is important.

### 3.4.1 Context Saved even in power loss

The context of the tasks is stored in RAM, SFRS and CPU registers, they are saved during a hibernate operation. So if a task is pre-empted and is in blocked state, its context is saved in RAM and if the power is lost, it can still regain its previous state just by restoring the RAM(before it comes into running state).

### 3.4.2 Energy-Aware Scheduler

The system navigates between the different LPM modes without loosing context, in fact a specific task before getting switched out of running mode is in a specific LPM mode and when it returns it can change LPM mode accordingly. The main reason why this is possible is because none of the memory controllers are being switched off in the LPM modes, this is a feature in Texas Instruments architectures. In ARM cortex-m Low power modes, there are multiple options to ensure which memory(RAM, ROM, caches) to power down. The highest priority of the system is assigned Hibernate/Restore, this cannot be avoided since they are the most critical tasks of the system.

But other tasks on the system can be modified to change their priorities based on which LPM they are currently in. ie, the tasks occupying lowest LPM would be assigned the highest priority so that they can occupy the running state more. Although implementing this is not an objective of this project, this is a good way to make an “energy-aware” scheduler.
CHAPTER FOUR: PERFORMANCE AND ANALYSIS

This chapter will detail the analysis of the performance of the RTOS, it will also detail parts of the RTOS that can be improved and methods to improve it.

4.1 Experimental Setup

The MSP430fr6989 board comes with an on board debugger and a special technology called EnergyTrace++[16] for Real-Time energy/power measurement with CPU and peripheral state monitoring. The board has a minimum supply requirement of 1.8Volts, maximum is 3.6Volts and a minimum current of approximately 10mA, considering the CPU + debugger circuit. Without the debugger circuit, the board runs at approximately 7mA. Fig 4.1 shows the experimental setup used to test the performance of the RTOS. Note: energytrace++ is a newer version of energytrace which has additional features.

![Experimental Setup Diagram](image)

**Fig 4.1 The experimental setup for testing Hibernus-RTOS.**

The diode is a schottky diode to make sure no negative voltages enter the MCU. The voltage divider is used because the comparator only has three pre-defined reference voltages which are 1.2V, 1.8V and 2.0 V. The 16µF capacitance is the board's in-built decoupling capacitance.

A function generator is used to mimic harvester outputs; I've also used a photovoltaic cell in place of the function generator to test the system. The photovoltaic cell gives 5V max at approximately 30mA under sunlight, it gives around 4.5V at 5mA indoors.
4.2 Analysis of the Startup of the RTOS

The following diagram shows an oscilloscope reading where the orange line monitors the GPIO pin P2.1(port 2, pin 12) of the MSP430fr6989 development board. The pin goes high when the execution enters the start of main() and goes low after all the tasks are created the same procedure is applied for the vTaskStartScheduler() function which starts the scheduler. The time the pin is HIGH indicates the time required for each operation. This shows that task creation takes approximately 8ms and scheduler start takes around 3.5ms.

![Oscilloscope reading illustrating the RTOS startup time after Reset](image)

The reading shows that if the task creation operation needs to happen during every restore then the system will not respond to highly intermittent voltages. So it is imperative to store task creation in FRAM. freeRTOS has left a process in place for this scenario but it is not complete. It doesn’t cover all the variables used in task create and it includes some variables.
used for the scheduler (which is not needed). Fig. 4.3 shows a part of the code that needs to be modified for this operation. The Macro PRIVILEGED_DATA can be expanded to the following:

```c
#define PRIVILEGED_DATA __attribute__((section(".fram_vars")))
```

This line will put the code shown in fig 4.3 in the DATA_FRAM | FRAM2 section. Selecting the required variables for saving task create into FRAM requires knowledge of the complete flow of the freeRTOS kernel code. Although this is very time consuming, it is not impossible and freeRTOS is one of the rare RTOSes that allows drastic changes to the kernel code, even to a point where it becomes something different.

```
374 PRIVILEGED_DATA TCB_t * volatile pxCurrentTCB = NULL;
375
376 /* Lists for ready and blocked tasks. ------------------------*/
377 PRIVILEGED_DATA static List_t pxReadyTaskList[ configMAX_PRIORITIES ]; /* Prioritised ready tasks. */
378 PRIVILEGED_DATA static List_t xDelayedTaskList1; /* Delayed tasks. */
379 PRIVILEGED_DATA static List_t xDelayedTaskList2;
380 PRIVILEGED_DATA static List_t * volatile pxDelayedTaskList;
381 PRIVILEGED_DATA static List_t * volatile pxOverflowDelayedTaskList;
382 PRIVILEGED_DATA static List_t * volatile pxSuspendedTaskList;
383 PRIVILEGED_DATA static List_t * volatile pxSuspendedWaitingCleanup = ( UBaseType_t ) NULL;
384 #if ( INCLUDE_vTaskDelete == 1 )
385     PRIVILEGED_DATA static List_t xTaskWaitingTermination;
386     PRIVILEGED_DATA static volatile UBaseType_t uxDeletedTasksWaitingCleanup = ( UBaseType_t ) NULL;
387 #endif
388 #if ( INCLUDE_vTaskSuspend == 1 )
389     PRIVILEGED_DATA static List_t xSuspendedTaskList;
390 #endif
391 /* Other file private variables. ------------------------*/
392 PRIVILEGED_DATA static volatile UBaseType_t uxCurrentNumberOfTasks = ( UBaseType_t ) NULL;
393 PRIVILEGED_DATA static volatile TickType_t xTickCount = ( TickType_t ) NULL;
394 PRIVILEGED_DATA static volatile BaseType_t xtspReadyPriority = pdFALSE;
395 PRIVILEGED_DATA static volatile UBaseType_t xSchedulerRunning = 0;
396 PRIVILEGED_DATA static volatile UBaseType_t xPendingTicks = ( UBaseType_t ) NULL;
397 PRIVILEGED_DATA static volatile BaseType_t xPendingQueue = pdFALSE;
398 PRIVILEGED_DATA static volatile BaseType_t xNumOfOverflows = ( BaseType_t ) NULL;
399 PRIVILEGED_DATA static UBaseType_t uxFreeTaskNumber;
400 PRIVILEGED_DATA static volatile TickType_t xNextTaskUnblockTime = ( TickType_t ) NULL; /* Initialised to 0 */
401 PRIVILEGED_DATA static TaskHandle_t xIdleTaskHandle = NULL; /* Holds the handle */
```

Fig 4.3 code snippet of tasks.c (snippet using Code composer studio)

Another possible fix to the problem is statically allocating the memory that is used by tasks and moving that memory into FRAM during hibernation. At the moment all tasks have their own dynamically allocated memory. The major advantage of dynamic memory allocation
is less wastage of memory, the system will only use the memory it requires and this will also lead to less energy wastage.

freeRTOS by default uses dynamic allocation because this makes it more portable across different microcontroller architectures. Because static allocation will require the programmer to have intimate knowledge about the memory maps of the hardware platform he is using as well as provide his own functions for handling overflows.

4.3 Store and Restore operations

The following oscilloscope reading follows the same procedure as before with GPIO Port2, Pin2 going high at the start of the storing task before the “Task Notify” signal is sent i.e., when store task is unblocked, moved into running state and goes low at the time the store task goes back into blocked state, this indicates the total time required to complete one store. The store operation of the original Hibernus takes 2ms. Here it takes approximately .3ms. The board here runs at same FRAM read/write frequency as the original Hibernus board, which is 8Mhz.

![Oscilloscope reading](image-url)

**Fig 4.4 Oscilloscope reading of the “store” operation**
There are multiple contributing factors to the speed difference. They are as follows:

1. **Use of Direct Memory Access (DMA)**

   All MSP430 has a unified data and instruction bus, which means that the CPU cannot operate at the same time as the DMA (stealing cycles). But the MSP430fr6989 has a “repeated burst mode” which puts CPU at 30% capacity. Still, for low power applications the DMA can offload a tremendous amount of data transfer work from the CPU, allowing a much lower power system. It also allows higher throughput. The original Hibernus project makes use of user-defined pointers for transferring RAM in and out of FRAM. Using DMA cancels that, the hardware handles all transfers.

2. **Switching speed of the task**

   As explained in previous chapters, the task memory is already created and the only delay in executing the task is its switching from blocked into running state, this is faster than creating individual memory or stack frames for every “hibernate()” function call, inside original Hibernus.

3. **Not using an intermediate temporary Variable**

   Hibernus uses an 230 long integer array temporary register named registers[230] in the store/restore process and stores the address of the address of the SFRs in another register called gen[230] this is shown in Fig 4.5 as a code snippet.

   A faster approach would be saving the addresses of the SFRs as Macros (as they will not occupy actual memory) and using the C-library function memcpy for the actual transfers into and out of FRAM. This translates to the following macro definitions and C-code:

   ```c
   #define SFR_BACKUP_LOC (FRAM2_START + RAM_SIZE)
   #define SFR_LOC 0x04B0
   #define SFR_SIZE 0x06
   #define SFR_BACKUP_LOC1 (SFR_BACKUP_LOC + SFR_SIZE)
   #define SFR_LOC1 0x100
   #define SFR_SIZE1 0x04
   (and so on…….)

   memcpy((uint16_t*)SFR_LOC1,(uint16_t*)SFR_BACKUP_LOC1,SFR_SIZE1);
   memcpy((uint16_t*)SFR_LOC2,(uint16_t*)SFR_BACKUP_LOC2,SFR_SIZE2);
   memcpy((uint16_t*)SFR_LOC3,(uint16_t*)SFR_BACKUP_LOC3,SFR_SIZE3);
   memcpy((uint16_t*)SFR_LOC4,(uint16_t*)SFR_BACKUP_LOC4,SFR_SIZE4);
   memcpy((uint16_t*)SFR_LOC5,(uint16_t*)SFR_BACKUP_LOC5,SFR_SIZE5);
   memcpy((uint16_t*)SFR_LOC6,(uint16_t*)SFR_BACKUP_LOC6,SFR_SIZE6);
   memcpy((uint16_t*)SFR_LOC7,(uint16_t*)SFR_BACKUP_LOC7,SFR_SIZE7);
   memcpy((uint16_t*)SFR_LOC8,(uint16_t*)SFR_BACKUP_LOC8,SFR_SIZE8);
   memcpy((uint16_t*)SFR_LOC9,(uint16_t*)SFR_BACKUP_LOC9,SFR_SIZE9);
   ```
memcpy((uint16_t*)SFR_LOC10,(uint16_t*)SFR_BACKUP_LOC10,SFR_SIZE10);
memcpy((uint16_t*)SFR_LOC11,(uint16_t*)SFR_BACKUP_LOC11,SFR_SIZE11);
memcpy((uint16_t*)SFR_LOC12,(uint16_t*)SFR_BACKUP_LOC12,SFR_SIZE13);

In the above code there is no intermediate variable used in the transfer process.

Fig 4.5 Code snippet of SFR store/restore from original Hibernus(snippet using notepad++)

4.4 Energy Trace++

The EnergyTrace tool show usage statistics of energy consumption by both hardware modules and software objects(like functions).
The following energytrace profiles are meant to show the maximum performance of the system without powerloss, so the voltage is at max Vcc of 3.6V. Energytrace circuitry only works through the board's USB connection. The possibility of using it through an external Texas Instruments debugger is not tested but if it does work then we can use energytrace to measure external supplies with the caveat that the external supply has to also power the additional debugger circuitry of the development board. Fig 4.6 showcases the input to signal to the setup. The green half sine wave is the oscilloscope reading at the voltage divider point of fig 4.1.

Fig 4.6 Oscilloscope reading of the input from signal generator to test maximum performance of RTOS at 6Hz

To experiment how the system deals with frequent requests for hibernate, the comparator module is reprogrammed to perform two Store operations at two thresholds as shown in the figure. The energytrace collects system data for 60 seconds.

Fig 4.7 and 4.8 shows the tests when AES is not running, so the only tasks running are hibernate, CRC and an update function for the Real Time Clock (RTC). The system clocks section
describes clock usage statistics between ACLK, MCLK, SMCLK etc. the MODOSC is a low power substitute for the source of MCLK at LPM1 and LPM2, this project does not make use of this. As expected ACLK is used the most because the tick generator of the RTOS is sourced by the ACLK. Using the input signal and the comparator setting mentioned in this section, hibernate will be performed 720 times per minute.

The purpose of the test is to see the effect of hibernate function being performed at a high frequency. Figure 4.7 shows that the system only occupies “active mode” for 1.5% and the rest of the time is in one of LPM modes. The CRC task consumes more energy than the hibernate task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Runtime (%)</th>
<th>Energy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low Power Mode</td>
<td>98.5</td>
<td>98.8</td>
</tr>
<tr>
<td>LPM3</td>
<td>97.9</td>
<td>98.2</td>
</tr>
<tr>
<td>LPM0</td>
<td>0.6</td>
<td>0.6</td>
</tr>
<tr>
<td>LPM2</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>&lt;Undetermined&gt;</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Active Mode</td>
<td>1.5</td>
<td>1.2</td>
</tr>
<tr>
<td>CCITT</td>
<td>0.5</td>
<td>0.4</td>
</tr>
<tr>
<td>CRC</td>
<td>0.3</td>
<td>0.2</td>
</tr>
<tr>
<td>memcpy</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>uxListRemove</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>xTaskIncrementTick</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>Store1</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>vTaskSwitchContext</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>vListInsertEnd</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>vPortYield</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>vPortPreemptiveTickISR</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>uiTaskNotifyTake</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>vTaskNotifyGiveFromISR</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Comp</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>vApplicationIdleHook</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>vTaskSuspendAll</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Fig 4.7 Energy trace++ profile of system usage statistics at 5Hz input to the Circuit

Both FRAM and DMA contribute very little to the overall energy consumption. Fig 4.9 shows the total energy/power consumed and even gives a simulated battery life estimate.

Note: Fig 4.7,4.8 and 4.9 are for a single test.
Fig 4.8 Energy trace++ energy profile showing peripheral usage at 5Hz input

Majority of the heavy lifting of the hibernation task is handled by the DMA module, because it transfers the biggest blocks of data in/out FRAM but as it is clear from fig 4.8 DMA accounts for very little energy consumption. Even memcpy which theoretically should be energy starving (it uses “register” qualifier) is not having much of an effect on sustainability.
The next test run is with the same properties except the AES tasks of encrypt and decrypt are run 20 times each. The energytrace is given in Fig 4.10.

---

**Fig 4.9 energytrace profile for the 5Hz signal**

**Fig 4.10 energytrace for the 5Hz signal with AES running**
From the figure, it is clear that even running AES tasks a few times in a minute has no significant bearing to the energy consumption of the RTOS.

Although a portion of the energy saving is from the scheduler system present in the RTOS, the power saving from using DMA and memcpy can be ported into the original Hibernus software, especially since it minimizes the energy consumption of the FRAM hardware as shown in fig4.8.
CHAPTER FIVE: PROJECT MANAGEMENT

5.1 Time Analysis

This chapter will discuss the proposed plan before the undertaking of any practical work and compare that with the actual final project.

Fig 5.1 Original proposed plan for the project as PERT chart

Fig 5.1 shows the proposed plan in the form of a PERT chart. The initial stage of the project involves:

- Research into transient algorithms and how they are implemented in software
- Research into RTOSes and more specifically into freeRTOS.

Rather than wait till the end of the time allotted for these segments, I started building the RTOS alongside reading up on it.

This also eliminated any risk of wasting time in the early stage, since the porting of freeRTOS into the specific hardware I had chosen ran into problems, they were:

- Modifying the linker file extensively.
- Porting the actual Hibernus c-code into the board I chose.
The experimental setup of the project was easier than expected and did not take a lot of time because the circuit was easy and used a simple sine wave from the signal generator for the initial testing of the hibernate operation.

The most time consuming process of the entire project happened to be debugging the code because Code Composer Studio (IDE used for development) does not have any provisions for debugging a running process, it can only debug a program from the start. This made debugging the “restore” operation very hard to understand since restore can only happen once power is cut off to the debugger hardware. One possible fix is to use GCC port of the MSP430 compiler in Linux and use gdb for debugging.

By porting normal Hibernus algorithm, I found out that the system was only doing a partial restore as explained in earlier chapters, the solution is to change a significant portion of the freeRTOS kernel files to make it a truly intermittent RTOS.

The final plan is given as the PERT chart in fig 5.2

![PERT chart](image)

**Fig 5.2 Final project plan**

**5.2 Cost analysis**

The only cost incurred is the cost of the development board which is 16.5GBP. All software used comes under GPL and is free to use.
CONCLUSION

This project required me to retrace a lot of the steps that went into the development of Hibernus and because of that, it gave me great insight into Transient Powered Computing, from proper usage of FRAM memory technology to context restoring techniques and dealing with energy harvester supplies. Programming with FRAM required going in depth into linker files and memory maps to see where individual variables which are written in a .c file is placed, this is also true for the process of porting the freeRTOS software into the MSP430 hardware. The implementation of the RTOS involved creating a multi-task, multi-interrupt RTOS. All of this deals with core embedded device development concepts and helped me to understand which embedded programming techniques is used in which situation. For example, what to use to create a critical section, which inter-task synchronization technique to use for a multi-task pre-emptive scheduler etc.

Although I have produced a prototype, the project is not finished mostly due to time constraints, the RTOS in its current state does not work for highly intermittent supplies. In the future I would incrementally change a lot of the core freeRTOS kernel code, perhaps to a point where it becomes a new RTOS. The reason is to convert a lot of the task parameters into FRAM variables and change some of the scheduler variables into FRAM variables as well. Doing so will ensure perfect Transient operation and since I was able to discover ways to improve the efficiency of the existing Hibernus code, the new RTOS will most definitely perform better than Hibernus. Further improvements like incorporating Hibernus++ functionality into the RTOS would be straightforward.
REFERENCES


