Performance Analysis of FreeRTOS and ChibiOS/RT Real-Time Operating Systems for Arduino Uno

1. Authors

Dapeng Lan <dapengl@kth.se>
Krishnaswamy Kannan <kkannan@kth.se>

2. Abstract

Real Time Operating System (RTOS) is an operating system which provides basic support for scheduling, resource management, synchronisation and communication with precise timing constraints. While RTOSes were predominantly employed in embedded systems with high-end 32-bit or 64-bit microprocessors, their usage in small systems with 16-bit or 8-bit microprocessors/microcontrollers has been gaining increasing popularity over the years. Recently, several RTOSes have been ported to the highly popular Arduino open source hardware platform thereby giving engineers and hobbyists an opportunity to learn about the fundamentals of a RTOS. This paper evaluates the performance of two such RTOSes namely FreeRTOS and ChibOS/RT on the Arduino Uno board with an 8-bit ATmega328P microcontroller. The real-time kernel functions such as context switching, interrupt handling, synchronization using semaphores, effectiveness of handling the priority inversion problem (predictability) were analysed and the memory usage and overheads incurred while using these core kernel functions were measured. The results obtained from our research illustrate that ChibOS/RT offers minimal latency for executing the above mentioned core kernel functions while FreeRTOS offers minimal memory usage with its compact kernel.

3. Introduction

3.1 Overview of Arduino

Arduino is an open-source electronic prototyping platform based on an easy-to-use hardware and software board that allows users to build interactive electronic objects. The Arduino boards house a microcontroller with several General

Purpose Input/Output (GPIO) pins, on-board storage, a USB connection, a power jack and support for programming the microcontroller with an Integrated Development Environment (IDE) called the “Arduino IDE”. These boards are available in different customized versions such as “Entry-Level”, “Enhanced features”, “Internet of Things”, “Wearable”, and “3D Printing”[1]. The Arduino boards have been gaining increasing popularity as they provide electronics hobbyists, students and engineers an inexpensive and easy way to create devices that interact with the environment using sensors and actuators [2].

3.2 Overview of FreeRTOS

FreeRTOS is a highly popular open-source RTOS from Real Time Engineers Ltd. with support for 35 microprocessor/microcontroller architectures [3] [4]. It is distributed under the GNU General Public License (GPL) with an exception that permits users' proprietary code to remain closed source while maintaining the kernel itself as open source, thereby facilitating the use of FreeRTOS in proprietary applications [5]. The FreeRTOS provides a small and simple RTOS kernel with a scheduler that supports round robin, pre-emptive and co-operative scheduling policies. It provides a set of easy to use Application Programming Interfaces (APIs) for creating threads with different priorities, software timers, inter process communication (IPC) by message passing and synchronisation between threads using mutexes or semaphores. It has a low overhead in performing kernel functions such as context switching, handling interrupts, a small ROM footprint and support for tick-less interrupt mode for low power applications.

3.3 Overview of ChibiOS/RT

ChibiOS/RT is an open source RTOS developed by Giovanni Di Sirio for deeply embedded applications with official support for 11 microprocessor architectures and unofficial support for 11 more microprocessor/microcontroller architectures [6] [7]. It is licensed under the GPL3 license with a minor exception for free commercial use that imposes restrictions in the number of deployments/license and further prevents the users from making any modifications to the commercially available modules [8]. However, a low cost fully commercial license of ChibiOS/RT is available without any of the above mentioned restrictions. The ChibiOS/RT kernel is a small, portable, static architecture with a scheduler that supports round robin and pre-emptive scheduling policies. It provides a set of APIs for threads with different priorities, virtual timers, semaphores, mutexes, event flags, messages, mailboxes and I/O queues. The kernel also provides some advanced features such as system state checker, cycle-accurate statistics, high resolution time and tick-less interrupt mode.

4. Theoretical Framework/Literature Study

In general, an operating system (OS) serves as the interface between the user and a computing device like a Personal Computer (PC), laptop or even a smartphone. It is a piece of software that manages the hardware resources in a computer and provides users with applications that enables them to interact with the computer. The OS running on PCs, laptops or smartphones are designed to provide the users with a set of system functions or applications that are not time critical. For instance, a file explorer crashing may not have a bad or a disastrous impact on the user. But, a delay or deadlock in processing signals in the video decoding system of a High Definition Television (HDTV) may spoil the quality of the video output and hence, such an application requires a real-time response. The class of operating systems which are designed to provide a time critical and predictable behaviour are called Real-Time Operating Systems (RTOSes). These operating systems emphasize on meeting deadlines and avoiding deadlocks rather than offering high speed performance or sophisticated features.

The key features of a RTOS are small memory footprint, minimal interrupt latency, minimal context switching time, IPC by message passing, synchronisation through mutex or semaphores, ability to resolve priority conflicts and support for masking interrupts. Some of the most popular RTOSes are VxWorks, Neutrino, µC/OS-II, FreeRTOS, Windows CE and RTEMS. RTOSes have been around for decades and they were generally used in high end systems with 32-bit or 64-bit microprocessors. But, the advent of more lightweight and compact open source RTOSes like FreeRTOS, Trampoline RTOS (OSEK), µC/OS-II have led to their increased use in small systems with 8-bit or 16-bit microprocessors/microcontrollers. This opened a world of possibilities to the engineers, electronics hobbyists and start-up companies as it provided them with a compact and fast kernel that reduced complexity and enabled code reusability thereby leading to a widespread use in small and low cost embedded applications.

The analysis of RTOSes has been an active research area for years with a large number of publications dealing with benchmarking of RTOSes for high end systems. However, there have been very few publications related to analysis of RTOSes developed for small microcontrollers. We wanted to concentrate on this area and hence we started exploring the RTOS ports available for small microcontrollers. We chose to focus on popular open source hardware platforms like the Arduino mainly because of their widespread use among engineers and hobbyists and a highly active developer community. On research, we found that a few RTOSes like FreeRTOS, ChibiOS/RT, DuinOS, NilRTOS were recently ported to the Arduino platform and we wanted to develop a reference model by benchmarking them. Due to time limitations, we chose only two most commonly used RTOSes for Arduino and benchmarked them with the most basic Arduino board namely, the Arduino Uno. An Internet search for relevant literature returned many papers that dealt with the analysis of advanced RTOSes for high end systems like VxWorks, Real-Time (RT) Linux, QNX and a few results for analysis of smaller RTOSes for small systems like µITRON, µC/OS-II. We choose a paper from each of the above mentioned categories for our reference.

Benjamin [9] in his paper titled, *"Performance Analysis of VxWorks and RT Linux"* has evaluated the performance of two advanced RTOSes namely, the WindRiver VxWorks and RT Linux. He conducted experiments on a MPC8260 microprocessor to measure context switching time, interrupt latency, binary semaphore acquire time, communication delay using message queues and time taken by the OS to resolve priority inversion. Benjamin's approach to measure context switching time and interrupt latency are complicated. He has configured the RTOSes to round robin scheduling to trigger context switches and has used a hardware timer to generate periodic interrupts to measure interrupt latency. A simpler method was used our research and its explanation can be found in section 6.

Tran and Su-Lim [10] in their paper titled, *"Survey and performance evaluation of real-time operating systems (RTOS) for small microcontrollers"* have compared the features of about 15 RTOSes and they have benchmarked the performance of 4 RTOSes namely, µC/OS-II, EmbOS, µTKernel and µITRON. They have run tests on a 16-bit Renesas m16C/62P microcontroller and they have presented their results for context switch time, semaphore acquire/release time, interrupt latency and communication delay using message queues along with the memory footprints for each of the tests. They have done an extensive research about the advantages of RTOSes over General purpose operating systems and their adaptability to small microcontrollers. We felt that the idea of tabulating the APIs used in the test methods gives a clear picture to the readers. So, in our study, we present the APIs used for each of the tests along with code snippets to provide an even better understanding to the readers.

5. Research Questions

Recent ports of FreeRTOS and ChibiOS/RT have given engineers and hobbyists an opportunity to learn about the fundamentals of a RTOS. However, *"the developers lack details about the performance of these RTOSes and hence are unable to make an informed decision about the RTOS that suits their application or requirements. This project aims to provide detailed information about 5 key performance metrics, which might enable the developers to choose an appropriate RTOS for the Arduino Uno board".*

6. Performance Metrics

The following section contains a brief description of the performance metrics that were tested in our study.

6.1 Context Switch

Context switch refers to the process of saving the context (status of program counter, processor registers and stack) of a task being suspended and restoring the context of a task being resumed. During context switching, the active task is switched out of the Central Processing Unit (CPU) to enable the waiting task to run. Context switching is a fundamental feature of a multitasking operating system and forms a critical part of real-time applications which are usually implemented using multiple tasks of execution. The time taken for a context switch is of paramount importance as it reflects the responsiveness of the RTOS.

6.2 Interrupt Latency

Interrupt latency refers to the amount of time that elapses from the time an interrupt is generated to the time when the interrupt is serviced i.e. the first instruction in the interrupt handler begins execution. The interrupt latency also includes the time taken for a context switch to the Interrupt Service Routine (ISR) or Interrupt handler. Since most embedded systems are interrupt-driven, the knowledge of raw interrupt latency of the RTOSes under consideration can give developers a clear picture of the minimum delay that will be incurred in processing interrupts in their design.

6.3 Semaphore acquire time

The synchronisation between tasks is facilitated by means of resolving resource conflicts through the use of mutexes or semaphores. A semaphore is a variable whose value indicates the status of a shared resource. Its purpose is to regulate the access to shared resources. Both RTOSes provide support for binary and counting semaphores. As most of the embedded systems using a RTOS have tight resource constraints, the synchronisation overhead incurred while using semaphores will significantly affect the application performance with increasing complexity.

6.4 Memory footprint

Small microcontrollers like the Arduino Uno have limited on-chip memory in order to keep the costs low. The amount of RAM storage available for programs is limited to 2048 bytes. So, RTOSes with large memory footprints are not suitable. Since, FreeRTOS and ChibiOS/RT advertise the fact that they are compact and not RAM hungry, the certainty of that claim is verified in our study to find their suitability to small microcontrollers and determine their raw RAM usage.

6.5 Priority inversion

Priority inversion is a scenario that occurs in a multitasking environment when a high priority task waiting for a shared resource is indirectly blocked by a low priority task that has locked the shared resource. This violates the fundamental principle that a high priority task can only be blocked by a task with a higher priority than that task. There are multiple solutions to this problem like priority inheritance, priority ceiling. Both, FreeRTOS and ChibiOS/RT use the priority inheritance mechanism to overcome this problem, where the priorities of the tasks under consideration are inversed when such a problem arises i.e. the low priority task which has locked the shared resource is assigned a high priority until it unlocks the resource. The time taken to resolve the priority inversion problem reflects the predictability of the RTOS and will give a measure of how fast the RTOS can resolve conflicts and deadlocks.

7. Hypothesis

Since, FreeRTOS is highly popular and relatively older than ChibiOS/RT, we expect FreeRTOS to outperform the ChibiOS/RT in all of the 5 key performance metrics.???

8. Benchmarking Methods

This section describes the benchmarking setup and test methods that were used to measure the performance metrics along with the results obtained from our study.

8.1 Benchmarking Setup

The hardware setup and IDE used for benchmarking the 5 performance metrics are described below:

- Arduino Uno board with ATmega328P microcontroller operating at 16MHz
- Arduino IDE 1.6.5 to run the test programs with FreeRTOS and ChibiOS/RT libraries
- Tektronix TDS 210 Oscilloscope to accurately measure latency.

8.2. Measurement Process

All the performance metrics under consideration deal with the measurement of the overhead incurred in performing kernel functions. So, we decided against using timer functions provided by Arduino libraries like the micros() for time measurement as they introduce an additional delay for their execution and eventually affect the accuracy of the results. However, the use of an external device like a logic analyzer or an oscilloscope will significantly improve the accuracy of raw latency measurements. Hence, we used a simple method of setting the Light Emitting Diodes (LEDs) ON (set) before starting the execution of the kernel function such as context switch or generating an interrupt and setting them OFF (reset) at the end of their execution. The LED's output was connected to a channel probe (CH1) of the oscilloscope. The time taken to set an LED ON/OFF was measured by initially setting and resetting the LED once and it was found to be 5µs (microseconds). The measurements for each of the performance metrics was obtained by deducting this time from the values obtained from the experiments.

All the test methods require the creation of threads and each RTOS provides its own API to create them. They are are described in the table below;

RTOS	Create Task API
FreeRTOS	chThdCreateStatic(waTask1, sizeof(waTask1), NORMALPRIO+2, Thread1, NULL);
ChibiOS/RT	xTaskCreate(Task1, "Task1", configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 2, NULL);

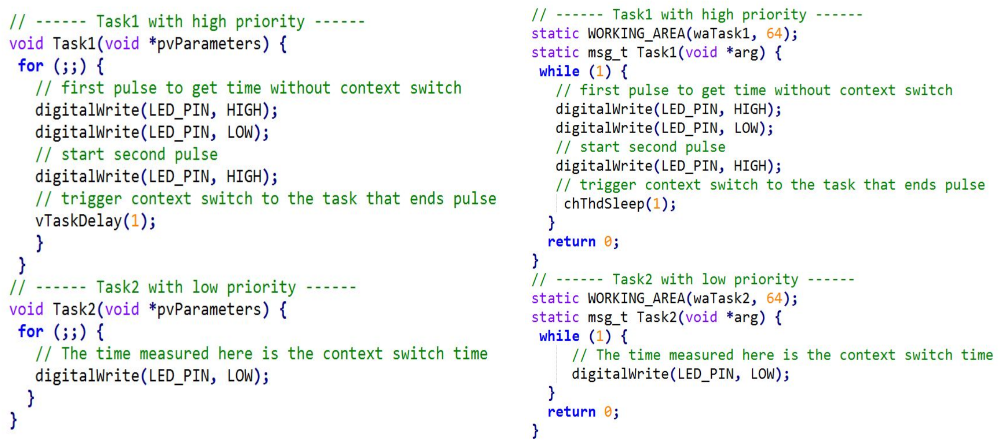
Table 1: APIs for Creating Tasks in FreeRTOS and ChibiOS/RT

The test methods along with the experimental results for each of the performance metrics are described below;

- **Context switching time**: Two tasks, one with high priority and another with low priority were created. The Task1 with high priority starts first and waits for a semaphore. So, the control moves to Task2 where the LED is set following which it signals/gives the semaphore to give back control to Task1. Once the Task1 resumes execution, the LED is reset. The context switching time was obtained by measuring the time between the point when the LED was set before giving the semaphore and the point when the LED was reset inside Task1.

RTOS	Context Switch APIs
FreeRTOS	vTaskDelay(<time >)
ChibiOS/RT	chThdSleep(<time>) (or) chThdYield()

Table 2: APIs for Context switch in FreeRTOS and ChibiOS/RT



```
// ------ Task1 with high priority ------
void Task1(void *pvParameters) {
 for (;;) {
   // first pulse to get time without context switch
   digitalWrite(LED_PIN, HIGH);
   digitalWrite(LED_PIN, LOW);
   // start second pulse
   digitalWrite(LED_PIN, HIGH);
   // trigger context switch to the task that ends pulse
   vTaskDelay(1);
   }
 }
// ------ Task2 with low priority ------
void Task2(void *pvParameters) {
 for (;;) {
   // The time measured here is the context switch time
   digitalWrite(LED_PIN, LOW);
  }
}
```

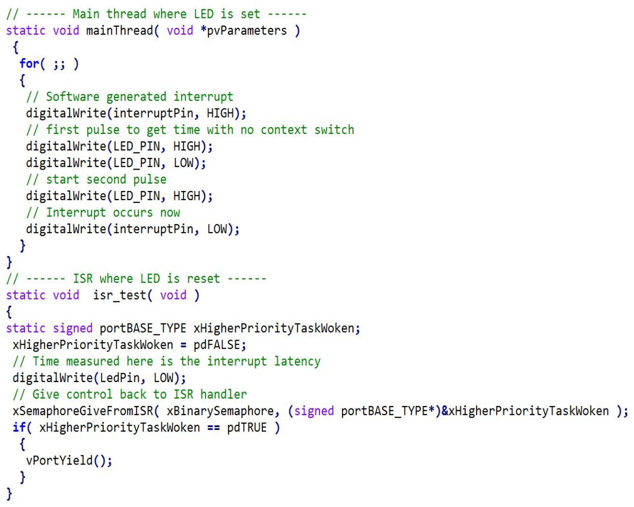
```
// ------ Task1 with high priority ------
static WORKING_AREA(waTask1, 64);
static msg_t Task1(void *arg) {
 while (1) {
   // first pulse to get time without context switch
   digitalWrite(LED_PIN, HIGH);
   digitalWrite(LED_PIN, LOW);
   // start second pulse
   digitalWrite(LED_PIN, HIGH);
   // trigger context switch to the task that ends pulse
    chThdSleep(1);
  }
  return 0;
}
// ------ Task2 with low priority ------
static WORKING_AREA(waTask2, 64);
static msg_t Task2(void *arg) {
 while (1) {
     // The time measured here is the context switch time
     digitalWrite(LED_PIN, LOW);
  }
  return 0;
}
```

Figure 1: Context switch time measurement for FreeRTOS (left) & ChibiOS/RT (right)

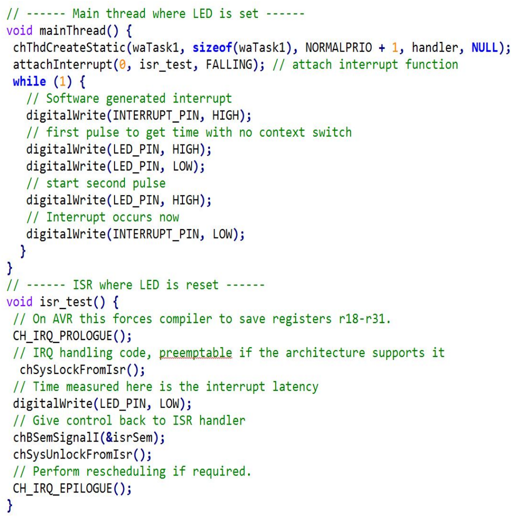
- **Interrupt latency**: A software interrupt was created in the main function by generating an interrupt on the falling edge of an interrupt pin. An LED is set in the main function before generating the interrupt. So, when an interrupt occurs the context switches to the interrupt handler or ISR where the LED is reset. The time between the point when the LED was set and the point when the LED was reset was measured as the interrupt latency. Note that the interrupt triggers a context switch to the interrupt handler and hence it also includes the context switch latency.

RTOS	Interrupt Handler APIs
FreeRTOS	attachInterrupt(0, isr_test, FALLING), xSemaphoreGiveFromISR()
ChibiOS/RT	attachInterrupt(0, isr_test, FALLING), chSysLockFromIsr(),chSysUnlockFromIsr(),

Table 3: APIs for Interrupt handler in FreeRTOS and ChibiOS/RT



```
// ------ Main thread where LED is set ------
static void mainThread( void *pvParameters )
 {
  for( ;; )
  {
   // Software generated interrupt
   digitalWrite(interruptPin, HIGH);
   // first pulse to get time with no context switch
   digitalWrite(LED_PIN, HIGH);
   digitalWrite(LED_PIN, LOW);
   // start second pulse
   digitalWrite(LED_PIN, HIGH);
   // Interrupt occurs now
   digitalWrite(interruptPin, LOW);
  }
}
// ------ ISR where LED is reset ------
static void  isr_test( void )
{
static signed portBASE_TYPE xHigherPriorityTaskWoken;
 xHigherPriorityTaskWoken = pdFALSE;
 // Time measured here is the interrupt latency
 digitalWrite(LedPin, LOW);
 // Give control back to ISR handler
 xSemaphoreGiveFromISR( xBinarySemaphore, (signed portBASE_TYPE*)&xHigherPriorityTaskWoken );
 if( xHigherPriorityTaskWoken == pdTRUE )
  {
   vPortYield();
  }
}
```



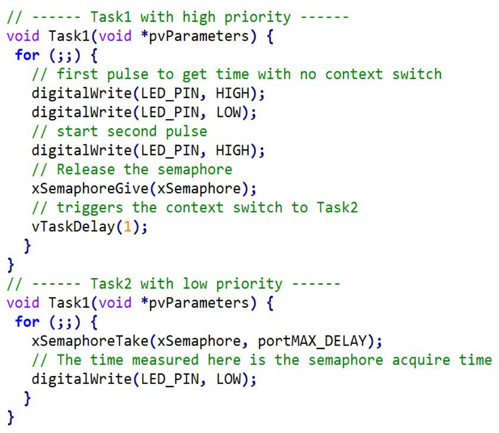
```
// ------ Main thread where LED is set ------
void mainThread() {
 chThdCreateStatic(waTask1, sizeof(waTask1), NORMALPRIO + 1, handler, NULL);
 attachInterrupt(0, isr_test, FALLING); // attach interrupt function
 while (1) {
   // Software generated interrupt
   digitalWrite(INTERRUPT_PIN, HIGH);
   // first pulse to get time with no context switch
   digitalWrite(LED_PIN, HIGH);
   digitalWrite(LED_PIN, LOW);
   // start second pulse
   digitalWrite(LED_PIN, HIGH);
   // Interrupt occurs now
   digitalWrite(INTERRUPT_PIN, LOW);
  }
}
// ------ ISR where LED is reset ------
void isr_test() {
 // On AVR this forces compiler to save registers r18-r31.
 CH_IRQ_PROLOGUE();
 // IRQ handling code, preemptable if the architecture supports it
  chSysLockFromIsr();
 // Time measured here is the interrupt latency
 digitalWrite(LED_PIN, LOW);
 // Give control back to ISR handler
 chBSemSignalI(&isrSem);
 chSysUnlockFromIsr();
 // Perform rescheduling if required.
 CH_IRQ_EPILOGUE();
}
```

Figure 2: ISR latency measurement for FreeRTOS (left) & ChibiOS/RT (right)

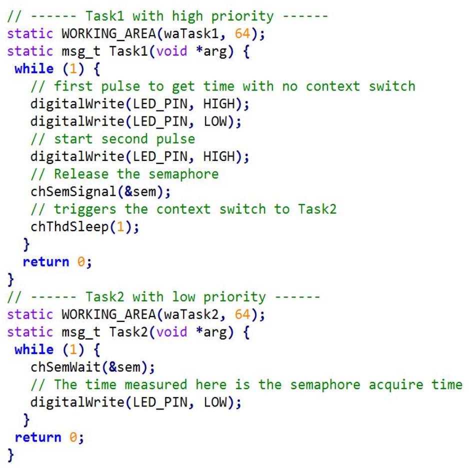
- **Memory footprint:** The RAM usage was measured from the Arduino IDE for each of the methods, after uploading the test code to the Arduino Uno board. The maximum available RAM in ATmega328P microcontroller on the Arduino Uno board is 2048 bytes.
- **Semaphore acquire time:** Two tasks, one with high priority and another with low priority were created. The Task1 with high priority begins execution first where an LED is set following which a semaphore is released and the Task1 goes to sleep. Now the context switches to Task2 where the semaphore is acquired and the LED is reset. The time taken by Task2 to acquire the semaphore was obtained by measuring the time between the point when the LED was set in before releasing the semaphore and the point when the LED was reset in Task2.

RTOS	Semaphore APIs (Sleep triggers context switch)
FreeRTOS	vSemaphoreCreateBinary(xSemaphore), xSemaphoreGive(<sem_name>), xSemaphoreTake(<sem_name, portMAX_DELAY)
ChibiOS/RT	chSemInit(&<sem_name>, <sem_value>), chSemSignal(&<sem_name>), chSemWait(&<sem_name>)

Table 4: APIs for using semaphores in FreeRTOS and ChibiOS/RT



```
// ------ Task1 with high priority ------
void Task1(void *pvParameters) {
 for (;;) {
   // first pulse to get time with no context switch
   digitalWrite(LED_PIN, HIGH);
   digitalWrite(LED_PIN, LOW);
   // start second pulse
   digitalWrite(LED_PIN, HIGH);
   // Release the semaphore
   xSemaphoreGive(xSemaphore);
   // triggers the context switch to Task2
   vTaskDelay(1);
  }
}
// ------ Task2 with low priority ------
void Task1(void *pvParameters) {
 for (;;) {
   xSemaphoreTake(xSemaphore, portMAX_DELAY);
   // The time measured here is the semaphore acquire time
   digitalWrite(LED_PIN, LOW);
  }
}
```



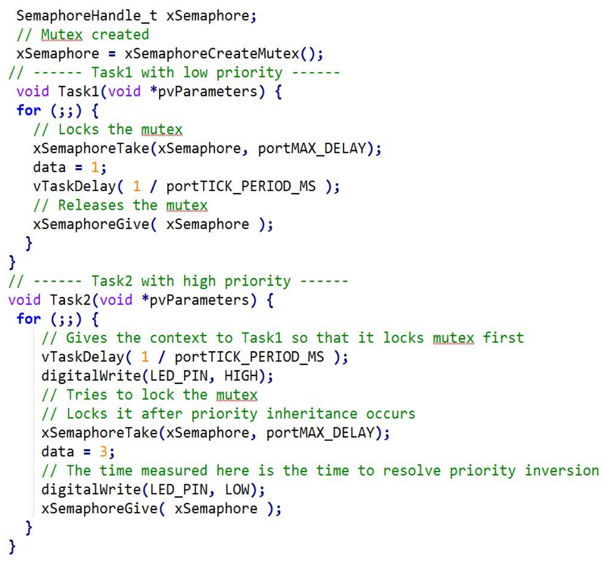
```
// ------ Task1 with high priority ------
static WORKING_AREA(waTask1, 64);
static msg_t Task1(void *arg) {
 while (1) {
   // first pulse to get time with no context switch
   digitalWrite(LED_PIN, HIGH);
   digitalWrite(LED_PIN, LOW);
   // start second pulse
   digitalWrite(LED_PIN, HIGH);
   // Release the semaphore
   chSemSignal(&sem);
   // triggers the context switch to Task2
   chThdSleep(1);
  }
  return 0;
}
// ------ Task2 with low priority ------
static WORKING_AREA(waTask2, 64);
static msg_t Task2(void *arg) {
 while (1) {
   chSemWait(&sem);
   // The time measured here is the semaphore acquire time
   digitalWrite(LED_PIN, LOW);
  }
 return 0;
}
```

Figure 3: Semaphore acquire time measurement for FreeRTOS (left) & ChibiOS/RT (right)

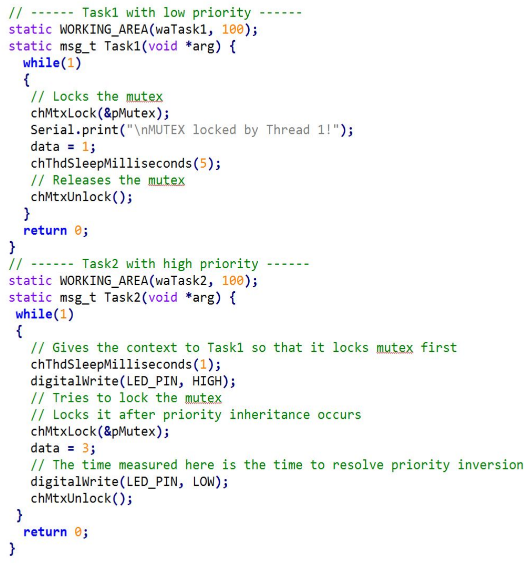
- **Time taken to resolve priority inversion:** Two tasks with different priorities were created. Task2 with high priority begins execution and hands over the context to Task1 with low priority. The Task1 locks the mutex for the shared resource and goes to sleep. Now, Task3 resumes and attempts to lock the mutex. It fails to the lock the mutex as it was already locked by Task1 and has to wait until it releases it. Since this results in priority conflicts, the Task1 is assigned the priority of Task2. So, the control shifts back to Task1 where the mutex is unlocked followed by which the priorities are restored. Finally, Task2 locks the mutex successfully and resets the LED. The time taken to resolve priority inversion was obtained by measuring the time between the point when the LED was set before Task2 attempts to lock the mutex and the point when the LED was reset after Task2 locks the mutex.

RTOS	**Mutex APIs (Sleep triggers context switch)**
FreeRTOS	xSemaphoreCreateMutex(),xSemaphoreGive(<sem_name>), xSemaphoreTake(<sem_name, portMAX_DELAY)
ChibiOS/RT	MUTEX_DECL(<mutex_name>), chMtxLock(&<mutex_name>), chMtxUnlock()

Table 5: APIs for using mutex in FreeRTOS and ChibiOS/RT



```
 SemaphoreHandle_t xSemaphore;
 // Mutex created
 xSemaphore = xSemaphoreCreateMutex();
// ------ Task1 with low priority ------
 void Task1(void *pvParameters) {
 for (;;) {
   // Locks the mutex
   xSemaphoreTake(xSemaphore, portMAX_DELAY);
   data = 1;
   vTaskDelay( 1 / portTICK_PERIOD_MS );
   // Releases the mutex
   xSemaphoreGive( xSemaphore );
  }
}
// ------ Task2 with high priority ------
void Task2(void *pvParameters) {
 for (;;) {
    // Gives the context to Task1 so that it locks mutex first
    vTaskDelay( 1 / portTICK_PERIOD_MS );
    digitalWrite(LED_PIN, HIGH);
    // Tries to lock the mutex
    // Locks it after priority inheritance occurs
    xSemaphoreTake(xSemaphore, portMAX_DELAY);
    data = 3;
    // The time measured here is the time to resolve priority inversion
    digitalWrite(LED_PIN, LOW);
    xSemaphoreGive( xSemaphore );
  }
}
```



```
// ------ Task1 with low priority ------
static WORKING_AREA(waTask1, 100);
static msg_t Task1(void *arg) {
  while(1)
  {
   // Locks the mutex
   chMtxLock(&pMutex);
   Serial.print("\nMUTEX locked by Thread 1!");
   data = 1;
   chThdSleepMilliseconds(5);
   // Releases the mutex
   chMtxUnlock();
  }
  return 0;
}
// ------ Task2 with high priority ------
static WORKING_AREA(waTask2, 100);
static msg_t Task2(void *arg) {
 while(1)
 {
   // Gives the context to Task1 so that it locks mutex first
   chThdSleepMilliseconds(1);
   digitalWrite(LED_PIN, HIGH);
   // Tries to lock the mutex
   // Locks it after priority inheritance occurs
   chMtxLock(&pMutex);
   data = 3;
   // The time measured here is the time to resolve priority inversion
   digitalWrite(LED_PIN, LOW);
   chMtxUnlock();
 }
  return 0;
}
```

Figure 4: Time measurement to resolve priority inversion for FreeRTOS (left) & ChibiOS/RT (right)

9. Results and Analysis

The results obtained for the context switching time, interrupt handling time, semaphore acquire time, time taken to resolve priority inversion by using priority inheritance mechanism and the memory footprint for all these test methods are shown in a tabulated form below;

	TIME MEASUREMENT			
RTOS	**Context switching**	**Interrupt Latency**	**Semaphore acquire**	**Priority inversion**

FreeRTOS	51 µs	58 µs (7 µs + 51 µs)	75 µs (24 µs + 51 µs)	101 ms
ChibiOS	16 µs	24 µs (8 µs + 16 µs)	27 µs (11 µs + 16 µs)	50 ms

Table 6: Time Measurements for FreeRTOS and ChibiOS

	RAM USAGE			
RTOS	**Context switching**	**Interrupt Latency**	**Semaphore acquire**	**Priority inversion**
FreeRTOS	325 bytes (15%)	350 bytes (17%)	325 bytes (15%)	385 bytes (18%)
ChibiOS	550 bytes (26%)	763 bytes (37%)	550 bytes (26%)	1,273 bytes (62%)

Table 7: Memory footprint of FreeRTOS and ChibiOS

The results obtained from our research disproved our hypothesis that "*FreeRTOS would outperform ChibiOS/RT in all 5 key performance metrics*". The highly popular FreeRTOS manages to outperform ChibiOS/RT only in the memory usage for the test code written for all the test methods. The ChibiOS/RT turns out to be a clear winner when it comes to the latencies in performing the core kernel functions such as context switching, interrupt handling, semaphore acquirement and offering better predictability in resolving deadlocks due to priority inversion. The FreeRTOS manages to offer better raw interrupt latency i.e. without taking context switching time to execute ISR into consideration. But in practice, this time is of great significance in affecting the performance. On analysing the data collected, we found that FreeRTOS could be suitable to large embedded applications due to its minimal memory usage while ChibiOS/RT is suitable for small applications which require a quick real-time response.

10. Discussion

In this paper, a simpler method was used to compare the five functions between FreeRTOS and ChibiOS/RT, context switching, interrupt handling, semaphore acquire time, priority inversion and memory footprint. As the analysis shows, the result disproved our hypothesis and ChibiOS/RT outperform FreeRTOS instead of the interrupt latency. However, ChibiOS use more RAM resource to achieve a better result, which means that FreeRTOS is more suitable for small systems with 8-bit or 16-bit microprocessors/microcontrollers.

For the future work, more embedded RTOS can be measured using the same measures we proposed, like DuinOS, NilRTOS, etc. Also our research result shows that potential improvement for the Embedded RTOS can be done. Keeping the low latency at the same time reduce the RAM usage as small systems with 8-bit or 16-bit microprocessors will become more popular as the trend of internet of things.

11. References

[1] Arduino Products, https://www.arduino.cc/en/Main/Products

[2] Waddington, N., Taylor, R. "Arduino & Open Source Design," 2007, http://www.sfu.ca/italiadesign/2007/page/papers/arduino-and-open-source-design.pdf

[3] FreeRTOS, http://www.freertos.org/RTOS.html

[4] EE Times' 2013 Embedded Survey, http://www.eetimes.com/document.asp?doc_id=1263083

[5] FreeRTOS License Details, http://www.freertos.org/a00114.html#exception

[6] ChibiOS/RT, http://chibios.sourceforge.net/docs3/rt/index.html

[7] ChibiOS/RT Supported Architectures, http://www.chibios.org/dokuwiki/doku.php?id=chibios:product:rt:architectures

[8] ChibiOS/RT License Details, http://www.chibios.org/dokuwiki/doku.php?id=chibios:licensing:start

[9] Empirical Research Methods, Carnegie Mellon University, http://oli.cmu.edu/courses/future/empirical-research-methods/