Technische Universiteit Eindhoven

Department of Mathematics and Computer Science
and Department of Electrical Engineering

Master's Thesis

Performance benchmarking of FreeRTOS and its Hardware Abstraction

By
Tao Xu

Supervisors

Prof. Dr. J.J. Lukkien
Dr. P.D.V. van der Stok
Dr. Ir. P.H.F.M Verhoeven
MSc. A.Schoofs
Dr. Ir. B.Mesman

November 2008

Keywords

FreeRTOS, hardware abstraction, benchmark, condition, performance

Abstract

This thesis presents a software benchmark tool to benchmark the performance of FreeRTOS and its hardware abstractions. The tool helps in gaining insight into the performance aspects of FreeRTOS and its hardware abstractions on a given hardware platform. The techniques and architecture of the benchmark tool are illustrated. Metrics benchmarked in the benchmark tool are discussed. Conditions influencing the performance were investigated in the benchmark tool.

Contents

List of Tables

List of Figures

List of Acronyms

BPS: Bits per second

BSN: Body Sensor Network

DCO: Internal Digitally Controlled Oscillator

GPIO: General Purpose Input/Output

I2C: Inter-Integrated Circuits

ISR: Interrupt Service Routine

SAND: Small Autonomous Network Devices

SPI: Serial Peripheral Interface

UART: Universal Asynchronous Receiver-Transmitter

WSN: Wireless Sensor Networks

List of Terminology

Benchmark: A benchmark is the act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it.

Branch: Branch is a point in a computer program where the flow of control can be changed.

Condition: In this thesis, condition is termed as values characterizing the environment of FreeRTOS and its hardware abstractions, especially parameters of functions, the number of tasks etc.

Configuration: In computing, configurations are actions to configure the initial settings for computer programs.

Context: In computer science, a task context (process, thread ...) is the minimal set of data used by this task that must be saved to allow a task interruption at a given time, and a continuation of this task at the point it has been interrupted and at an arbitrary future date.

Device: a device is a concrete module linked to the micro-controller, a peripheral or via another device. A device can be a sensor or any other chip linked to the micro-controller. E.g.: ADC (linked via I2C bus), accelerometer sensor (via internal or external ADC), radio chip (via SPI bus), and terminal (via UART bus) . . .

Function pair: A couple of functions which are commonly used as a set, with or without any code in between, in order to do a specific job. E.g. the beginning function of a given timing measurement and the end function of the given timing measurement.

Hardware abstractions: Sets of routines in software that emulate some platform-specific details, giving programs direct access to the hardware resources.

Hardware platform: The processor and its peripherals placed on a board.

Interface: An interface is the communication boundary between two layers of the peripheral driver architecture. It realizes the abstraction that a layer provides of itself to the outside. It has to be understood as in API (Application Programming Interface). An interface is defined by a set of functions allowing the upper layer to access the functionalities of this lower layer.

Micro-kernel: A microkernel is a minimum computer operating system kernel which, in its purest form, provides no operating-system services at all, only the mechanisms needed to implement such services, such as low-level address space management, thread management, and inter-process communication (IPC).

Mini-kernel: Mini-kernel is termed as the core of a kernel, which provides facilities including process management, memory management and system calls.

Peripheral: A peripheral is “constituting an outer boundary” of the microcontroller chip. The peripherals are all the hardware interfaces between the processor and external devices including buses, GPIO and timers.

Primitive: Primitive is a system of instructions and data executed directly by a computer's central processing unit.

Software platform: A kernel and its peripheral drivers to provide software developers a platform to enable a portable code style. In this report, software platform commonly represents the FreeRTOS kernel and its hardware abstractions.

Chapter 1

Introduction

The idea of ubiquitous computing inspired the scientists at Philips Research to propose the vision of Ambient Intelligence in 1999. "It is the vision of a world in which technology, in the form of small but powerful silicon chips, will be integrated into almost everything around microsecond, from where it will create an environment that is sensitive to the presence of people and responsive to their needs."(Philips 2008) Ambient intelligence is more than just a vision, though. It forms the basis of a large variety of current research programs at Philips Research and also on a European level. In 2001, Philips' vision of ambient intelligence was adopted as the leading theme for the Sixth Framework on Information Society and Technology (IST) Research in Europe, which resulted in a research program with a budget of 3.7 billion Euros over four years (Philips 2008). Typical application scenarios are not only seen in home environments, but also in health care and security systems.

1.1 FreeRTOS and its Hardware Abstraction

To ease field testing of applications for Ambient Intelligence, Philips Research Eindhoven came up with a number of sensor node platforms. Common problems arose on all those platforms. Application development was hard to achieve. The developers faced all the same difficulties: they had to deal with many basic hardware functionalities before they could focus on data processing and other high level algorithms.

To solve these problems, a kernel-based software environment in Philips Research was needed to achieve a common programming platform for a variety of different hardware platforms. After a study on available operating systems on the market (Catalano 2006), FreeRTOS, a small and very basic RTOS was proposed to form the basis of this programming platform and a portable driver architecture was added to it. Later, high-level operating system services and applications could be implemented on top of this including time and task synchronization (Aoun 2007), 802.15.4 MAC Layer (Schoofs 2006) etc.

1.2 Motivation and Objectives

With an available kernel and hardware abstraction, software developers gain great advantages. Application code can be developed to be portable and flexible. Software developers can focus on application level of programming without studying the low-level details of hardware. The work focuses on the applications and the burden for high-level programs is reduced. But at the same time, software overhead of the FreeRTOS kernel and its hardware abstraction is not negligible. The FreeRTOS kernel and its hardware abstraction create a software layer and a set of system facilities, which increases latency and resource overhead. As a result, users of the FreeRTOS kernel and its hardware abstraction need information of its overhead and performance, which is at the base of the benchmarking initiative.

There are a number of aspects affecting the overhead and the performance. After the FreeRTOS kernel and its hardware abstraction were developed, it was ported onto three sensor node platforms in Philips and demos were made. To better study the demo behaviors, two metrics were benchmarked (Catalano 2008) (Preusker 2008). It was found that not only individual system calls had their own execution time but conditions of functions could also affect the results. Due to the time limitation, only two metrics of FreeRTOS were investigated. But it was suspected that more conditions would affect the performance. To investigate dependencies between the performance and conditions, an investigation of conditions is needed besides a performance benchmark.

In the context of methodology, in the literature (Catalano 2008; Preusker 2008), measurement routines were developed and run individually for different metrics. Measurement programs were inserted into multiple segments of files which did not form a standard suite of programs. In addition, metrics were measured by an oscilloscope and results of measurements were shown in a non-intuitive way. Measurements were non-reproducible and newcomers had difficulties in carrying out a benchmark on a new instance of hardware platform. Inspired by this problem, the idea of a configurable benchmark tool was originated. The benchmark tool is configurable to different hardware platforms and the measurement of a number of metrics.

Given the problems stated above, this thesis project has following objectives:

- Identify metrics for benchmark and measure them
- Develop a configurable benchmark tool for FreeRTOS and its hardware abstraction
- Identify conditions influencing the performance of the FreeRTOS kernel and its hardware abstraction by using the benchmark tool

1.3 Performance Benchmark

A benchmark is the act of running a computer program, a set of programs, or other operations, in order to assess the performance of an object, normally by running a number of standard tests and trials against it. The term 'benchmark' is also utilized for the purposes of elaborately-designed benchmarking programs themselves.(Wikipedia 2008)

Performance benchmarking in this thesis is the act of running a benchmarking program, which is call the benchmark tool, to evaluate the performance of the FreeRTOS kernel and its hardware abstraction. The performance benchmarking is carried out by running the benchmark tool a number of times.

1.4 Benchmarking scope

In accordance with the topics of research programs within the ambient intelligence vision, FreeRTOS and its hardware abstraction are designed and targeted to different application scenarios. FreeRTOS and its hardware abstraction are applied to a number of functional fields. The applications built on top of them can vary dramatically. As a result, the workload conditions can not be explicitly defined. This does not satisfy the fundamental requirements for an application-level benchmark. This made a benchmark with a use case an unviable approach in this thesis work. Instead, benchmarking the unloaded mini-kernel and hardware abstraction gives clearer perspectives of the behavior of the FreeRTOS kernel and its hardware abstraction so it is more attractive to developers. Inspired by these specifications, the benchmarking in this thesis concentrated on functions in the mini-kernel and hardware abstraction.

1.5 Outline of Thesis

This thesis is structured as follows. In Chapter 2, the real time mini-kernel FreeRTOS and its hardware abstraction are described. In Chapter 3, metrics within the benchmark scope are outlined. Chapter 4 introduces the benchmark, with its requirements, techniques and attributes. In Chapter 5, the proposed benchmark tool is described. In Chapter 6, the benchmark tool will be used on two sensor node platforms as case studies to benchmark the FreeRTOS and its hardware abstraction. In Chapter 7, performance analysis is made as a conclusion.

1.6 Related work

To start with, a set of system calls were measured at Philips Research (Catalano 2008) (Preusker 2008), which stimulated this thesis work.

Inspired by these exercises and motivated by objectives in section 1.2, related work includes the following works: identifying metrics for benchmark, exploring methodologies for benchmark and investigating existing tools for measurement.

First, the literature study explored a range of papers published about metrics. Performance analysis of real-time operating systems is a hot topic. A number of papers discussed kernels with new features and compared metrics with existing kernels. After investigation, metrics discussed in these papers could be divided into two categories:

A subset of metrics depends on user-level applications or scenarios (Baynes, Collins et al. 2001 ; Douglas C.Schmidt 2002; Cormac Duffy 2006; Olusola 2007). As mentioned in the benchmarking scope (section 1.4), applications and scenarios for the FreeRTOS kernel and its hardware abstraction are not defined, so none of these metrics are discussed in this thesis.

The other subset of metrics in the literature focused on application-independent measurements. Some of them did experiments on latencies of functions (Gopalakrishnan 2005; Park, Kim et al. 2006). Others studied relationship between metrics and kernel loads (Gopalakrishnan 2005). Because of the application-independent aspects of this thesis work, these gave desirable references for targeted metrics.

Next, methodologies used in the literature were studied. For experimental environments, most of them were based on a specific test bed (Douglas C.Schmidt 2002; Gopalakrishnan 2005; Cormac Duffy 2006; Park, Kim et al. 2006 ; Olusola 2007) or a high-level language model (Baynes, Collins et al. 2001). For methods gaining measurement results, a number of methods were used, such as estimation (Park, Kim et al. 2006), an object request broker (Douglas C.Schmidt 2002), compilation (Olusola 2007), system time stamping (Cormac Duffy 2006), high-level language model (Baynes, Collins et al. 2001) or kernel-support tools (Gopalakrishnan 2005). In addition, to estimate the duration of sequence programs, some literature (Puschner and Koza 1989; Mueller 2000; Antoine Colin 2001) provided methods to calculate worst case or maximum execution time.

One of the main objectives is to create a configurable benchmark tool for FreeRTOS and its hardware abstraction. Existing measurement tools were investigated and their possibilities of being integrated to the objective measurement tool were explored. The existing measurement tools for profiling fall into following categories:

- Tools supporting widely used kernels, such as Imbench (Larry McVoy 1996), Linux trace toolkit and Oprofile . FreeRTOS is not on the list.
- General analysis tools for code examinations, such as Gprof and HEPTANE(IRISA 2003). They are only applicable to general purpose computer

platforms, so they do not fit in the general embedded systems environment of benchmarking scope.

- Analysis tools existing on their own specific hardware platforms, such as SPYDER-CORE-P1 (Weiss, Steckstor et al. 1999) and ATOM (Eustace and Srivastava 1995). Most Philips specific sensor nodes in the benchmarking scope are hardware which is not included in the hardware list of those analysis tools.
- Trace utility tool (Barry 2008) supported by FreeRTOS. It provides run-time scheduling information and high water mark of the stack of each application task. It stays in application level, which is designed for debugging but not for benchmarking.

In conclusion, the literature gives a list of metrics for benchmarking, which is developed further in Chapter 3. The literature also provides a set of methodologies for measurement. Together with the available programming environment, a set of methodologies was selected and proposed in section 4.4. For the profiling tools, unfortunately, due to the particularity of FreeRTOS and its hardware abstraction, existing tools are not applicable in this thesis work.

Chapter 2

FreeRTOS and its Hardware Abstraction

This chapter deals with FreeRTOS, an open source mini real time kernel, and its hardware abstraction developed by Philips Research on top of FreeRTOS. In the first part, main features of FreeRTOS are presented after a short introduction. The Application Programming Interface (API) is described in Appendix A. In the second part, the hardware abstraction is described. Its architecture and individual modules are illustrated. Its API is described in Appendix B. The content of this chapter is mainly based on the FreeRTOS official site (Barry 2008) and the technical report of Philips Research (Catalano 2008).

2.1 FreeRTOS

FreeRTOS is a portable, open source, mini Real Time Kernel. It was ported to fourteen hardware architectures from 8-bit small micro-controllers to full featured 32-bit processors including ARM7, ARM9, MSP430, AVR, PIC and 8051 (Barry 2008).

FreeRTOS is portable. The porting is eased by several factors. First the FreeRTOS code base is small. It composes a total of three core files and an additional port file needed for the kernel itself. Secondly FreeRTOS is mostly written in standard C. Only a few lines of assembly code are necessary to adapt it to a given platform. Finally FreeRTOS is heavily documented in the source code as well as on the official website(Barry 2008)

with an application-level benchmark (Barry 2008). FreeRTOS is open source. It is licensed under a modified GPL and can be used in commercial applications under this license. FreeRTOS code is freely available on its website, which makes the kernel easy to study and understand.

The following sections describe FreeRTOS main features. More details can be found at (Barry 2008).

2.1.1 Scheduler

FreeRTOS features a Round Robin, priority-based scheduler. Each task is assigned a priority. Tasks with the same priority share the CPU time in a Round Robin fashion.

The FreeRTOS scheduler can be configured as preemptive or collaborative. The real time behavior of the system requires preemptive scheduling. For simpler systems, collaborative scheduling can be used.

The preemptive scheduler can stop a running task during its execution (preempt the task) to give CPU resources to another ready task. This feature is used for CPU time sharing between ready tasks with the same priority. It is also used in case of an interrupt which may wake up a task waiting for a signal or for some data. The woken task should have a higher priority than the current running task to be allocated CPU time directly.

2.1.2 Inter-task Communication

FreeRTOS provides several methods for inter-task communication, including message queues and binary semaphores. FreeRTOS queue mechanism can be used in the communications between two tasks and the communication between tasks and Interrupt Service Routine. A queue is a structure able to store and restore data.

Semaphores in FreeRTOS are actually implemented as a special case of queues. A semaphore is a queue of one single element with size zero. Semaphore *take* operation is equivalent to a queue *receive*, whereas semaphore *release* (or *give*) operation is equivalent to queue *send*. Note that on initialization, the semaphore queue is full. Semaphores are used for task synchronization and mutual exclusion. A section of the application can be protected by a semaphore to enable mutual exclusion. The first task executing a section of mutual code takes the semaphore. Any other task willing to execute this code will wait on the semaphore until the first task releases it.

A task willing to receive a byte from an empty queue, to send a byte to a full queue or to take an already taken semaphore will be blocked by the kernel. A task decides the maximum time it allows the kernel to block it in the inter task communication system call parameter. When the semaphore or the queue becomes available again, the kernel will ready the task. It will be allowed to run if it has the highest priority. In the case the semaphore or the queue stays busy and the waiting time has elapsed, the kernel will ready the task again, and return an error to it. It is the task's responsibility to check this return value.

2.1.3 Memory Management

The RTOS kernel has to allocate RAM each time a task, queue or semaphore is created. Three RAM allocation schemes are included in the FreeRTOS.(Barry 2008)

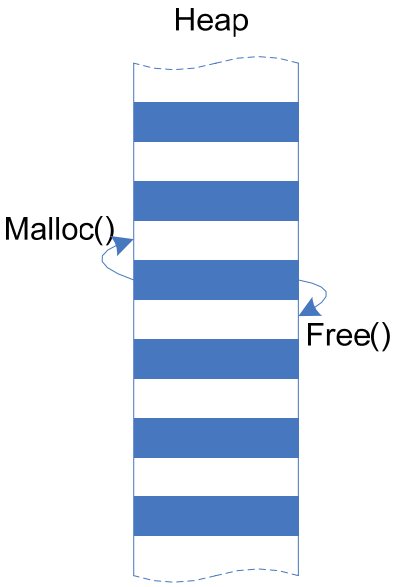


Figure 1 Memory management

The first scheme reserves a huge table in the memory called the heap. Every time the memory allocation system call is used (*malloc*), the pointer for free space is incremented with the allocation size and a pointer is returned to the application. The memory freeing (*free*) simply does nothing.

FreeRTOS provides nevertheless a more complex scheme for memory allocation. The second scheme uses the best fit algorithm to re-allocate memory blocks which have been freed. This allocation scheme is not deterministic and the real time behavior of the system can be affected.

Finally, a third scheme is provided which uses the standard library *malloc()* and *free()* functions. It is not deterministic as well, needs support from the compiler and might induce a lot of memory overhead.

In the benchmarking scope (section 1.4), a number of applications is loaded with periodic tasks and has restrictions on power consumption. The first memory management scheme was chosen. It is used to avoid the hassle of the garbage collection algorithm and to keep things simple. The rationale is that most of the applications

targeted to such small devices will never use memory freeing anyway and that garbage collection affects the real-time behavior of the system. Thus the first scheme achieves a low power consumption and real time behavior.

2.1.4 FreeRTOS data structures

A set of structures defined by FreeRTOS are discussed in the section. The data structures of tasks, queues and lists are illustrated.

The Task Control Block (TCB) of tasks includes various members to keep track of the stack state, the task status and information about the task. The stack state is controlled by two pointers, one pointing to the top of the stack and one pointing to the beginning of the stack. Task information is kept in the TCB. Task name, priority, TCB number are all stored as integer or characters values in the TCB. Task status is stored in the TCB in the form of two list items (lists are described further in this section). A task can therefore be part of two lists, a generic list when the task is ready or delayed and an event list when the task is blocked. For example, when a task is blocked on a semaphore, the task's event list item will be inserted in the semaphore waiting-to-take list and the generic list item will be removed from the ready list and inserted in the delayed task list with the delay value as specified in the semaphore take function call.

A queue is materialized in the heap by a control structure and a data storage space. Four pointers manage the queue state, pointing respectively at the beginning, the end of the queue, at the next byte to write to and the next byte to read from the queue data. The queue's status is kept into five integer values at the end of the queue control block, the number of messages waiting to be read, the total queue length in number of items, the item size, the number of items read and wrote while the queue is locked. Finally, the queue structure holds two lists, one for tasks waiting to send items to the queue and the other for tasks waiting to receive items from the queue while the queue is respectively full or empty.

A semaphore is, in FreeRTOS, a special case of a queue. As explained earlier, taking a semaphore is equivalent to receive from a queue, thus a task waiting to take a semaphore will be stored in the task waiting to receive list, renamed here task waiting to take list. A semaphore being a queue of one element of size zero, the item size will be zero, the total queue length will be one and the number of messages waiting will be binary, either 1 if the queue is full i.e. the semaphore is free, or 0 when the queue is empty i.e. the semaphore is taken.

The last FreeRTOS structure is the list structure. A list is composed of a list structure and list items. The list structure holds the number of items in the list as an integer value and two pointers. One is pointing to the list-end item containing the maximum possible item value, meaning it is always at the end of the list. The other is pointing to the latest item accessed. A list item has an item value which is an integer used to sort the items in the list. It contains a number of pointers, pointing to the next and previous items which can be the list-end item if the item is at the end or at the beginning of the list. Another member of the structure points to the owner of the item, for instance a TCB and a last one points to the container of the item, namely the list structure.

2.2 Hardware abstraction

This section introduces the hardware abstraction developed on top of FreeRTOS. The purpose of the architecture is to define a standard hardware abstraction architecture to access hardware from applications. The benefits are numerous. Defining a standard hardware abstraction architecture allows faster application development, as the programmer does not have to deal with the hardware architecture, but can focus only on a well defined interface. It is optimized for the underlying hardware. All the hardware options are optimized and taken into account. It also enables a lot of code re-use between different project applications. It is developed for FreeRTOS and uses all the mechanisms available in the OS. Hardware interaction with the OS is then optimized.

The goal of this hardware abstraction architecture is to abstract completely the hardware mechanisms to the application. Application only reads and writes data from operating system structures, like queues and semaphores. The underlying software provides complete functions to deal with the hardware buses (such as I2C, SPI or UART) to physically send and receive data. This architecture should allow applications and device drivers to be stacked on top of it, and provide all the necessary functions in order to communicate fully with the different hardware blocks.

2.2.1 Multi-layer abstraction architecture

The multi-layer abstraction architecture was designed as shown in Figure 2. The different layers have very specific tasks to do, and provide a clear interface to the upper layer. The arrows connecting different layers illustrate the dependency of function calling. To be complete, this section will describe those interfaces. The layers are: basic functions layer, interrupt subroutines layer, hardware presentation layer and peripheral driver layer.

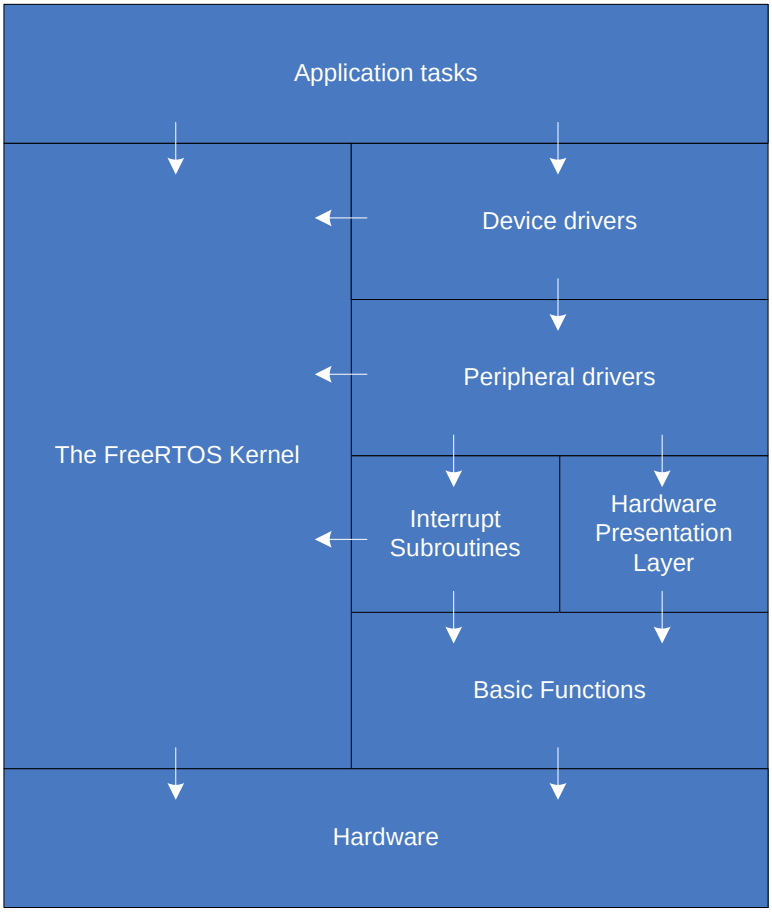


Figure 2 Hardware Abstraction Architecture

Basic Functions Layer

The basic functions layer is a very thin layer enabling a way to read and write to the input and output registers of the micro-controller and to redirect hardware interrupt to the interrupt subroutine layer.

These functions are used a lot in every peripheral driver, as they are the only means to communicate with the underlying hardware. Making standard functions allows the upper layers to use a fixed interface for every hardware transaction, which makes the code more readable and easier to debug.

In general, this layer is implemented as macros or inline functions. Instead of using the register addresses to address a certain peripheral register, the symbolic name (taken from the data-sheet) is used. It makes the code more readable, easier to port and to maintain.

Interrupt Subroutines Layer

The interrupt subroutines layer handles the interrupt working on the peripheral driver structures, like freeing a semaphore or posting to a queue, using the operating system interface, and operating on the hardware, such as acknowledging the interrupt or sending an extra byte.

The goal is to make standard interrupt subroutines, which will deal with the drivers for the peripherals. As micro-controllers have a finite number of peripherals and of interrupts linked to those peripherals, the ISR layer is closely connected with the peripheral driver layer. The interrupt subroutines layer have a subscription mechanism which allow application or device drivers to perform custom actions within the interrupt, in addition to standard interrupt actions predefined by the peripheral driver.

Hardware Presentation Layer

The hardware presentation layer (HPL) provides a complete and dense interface of all the hardware peripherals to the upper layer, while using only the thin basic functions layer.

The goal is to provide an easy to use interface, with explicit names and standard functionalities to the upper layers. This layer is stateless, meaning that it will not keep information about the status of a peripheral. It will execute straightforward the order from upper layers.

Peripheral Driver Layer

Finally, the peripheral driver layer provides to the application or to the upper device drivers a set of functions to use the hardware safely and in interaction with the OS. In this layer, the state of the interface is kept in an OS structure and the data and/or the events related to a bus or a timer are also recorded in OS queues or semaphores.

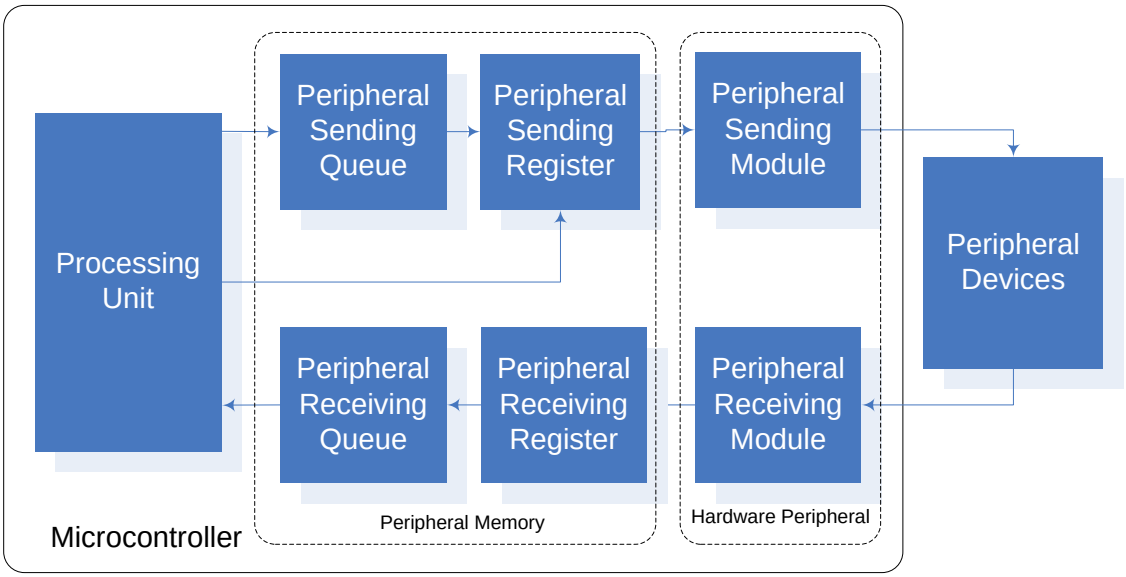


Figure 3 The hardware connection of peripherals

Figure 3 illustrates the hardware connection of peripherals for the microcontroller and the peripheral devices. A microcontroller contains three hardware components: a processing unit, a space of peripheral memory and hardware peripherals. The peripheral memory consists of the peripheral registers and the peripheral sending and receiving queues, created by the peripheral driver layer. The hardware peripheral consists of a sending module and a receiving module. Two channels are present in the hardware connection: a sending channel and a receiving channel. Both channels are maintained by the hardware abstraction. When a peripheral sending function is called, the hardware abstraction checks the status of the peripheral sending module. If the peripheral sending module is free, the data to be sent are pushed into the peripheral sending register. Otherwise, the data are pushed into the peripheral sending queue, where the data are transferred into the peripheral sending register by the interrupt subroutines. In the receiving channel, the data sent from the peripheral devices are received and transferred

to the peripheral receiving register by the peripheral receiving module. Then the data are pushed into the peripheral receiving queue by the interrupt subroutines.

Chapter 3

Metrics for Benchmark

This section outlines the metrics that are identified for benchmarking.

First, as the benchmark scope in section 1.4, the benchmark of the FreeRTOS kernel and its hardware abstraction in this thesis concentrates on the mini-kernel and the hardware abstraction. The related work in section 1.6 has listed the application-independent metrics within the benchmarking scope. These metrics consist of memory read/write time and task creation latency, context switch time, scheduling latency, system call overhead and average duration of interrupt disable time. All these metrics form the latencies of system calls of FreeRTOS and the peripheral functions. Second, communications between the processor and its peripherals are evaluated. The speed of data transmission is one of the metrics for the hardware abstraction, known as throughput. At last, as discussed in section 1.2, the resource overhead of the FreeRTOS kernel and its hardware abstraction needs to be benchmarked, which is included in the metrics list. The three sets of metrics, given above, form the list of metrics for this thesis work.

3.1 Latencies

The latency of an operation is termed as the amount of delay encountered when a given operating system function is executed. The latencies encountered are a good indication of the temporal behavior of the kernel. The temporal performance is one of the key aspects of the motivation section (section 1.2).

3.1.1 System calls latency

System call latency is defined as the amount of time lapsed between the moment a system call is made (to use some kernel facilities), and the moment the execution returns. This has to be collected for different widely-used system calls, like task handling, kernel control, queue management and semaphores management, etc. As illustrated in the hardware abstraction structure (Figure 2), system calls are the interfaces by which most of the functionalities of the operating system are exposed to application level. Their durations directly influence applications.

3.1.2 Latency of peripheral functions

As the definition of latency of system calls, the latency of peripheral functions is defined as the amount of time lapsed between the moment a peripheral function is invoked (to use some peripheral devices), and the moment the execution returns. This has to be collected for a number of peripheral functions within the hardware abstraction architecture, such as General Purpose Input/Output (GPIO), Universal Asynchronous Receiver-Transmitter (UART), Serial Peripheral Interface (SPI) and Inter-Integrated Circuits (I2C). As illustrated in the hardware abstraction architecture (Figure 2), peripheral functions are the interface by which the facilities of the peripheral drivers are exposed to application level. Hence, it is important that this delay is benchmarked.

As listed in Appendix B, the latencies might depend on the values of the parameters passed to the functions. A minimum set of parameters was involved in the measurement.

3.1.3 Context switch overhead

The context switch overhead is defined as the amount of time taken by the operating system to switch from one task to another, without any other task or interrupt subroutine being executed in between (Larry McVoy 1996). In FreeRTOS, the

procedure of context switch is visible in the code of tick timer interrupt, which consists of two parts excluding the context saving and restoring:

- Increase the tick count and check if any task that is blocked for a finite period, requires its removal from a blocked list and placing on a ready list
- Set the pointer to the current Task Control Block (TCB) to the TCB of the highest priority task that is ready to run.

3.2 Throughput

Throughput of a system is the rate at which a particular system can move (or process) data. Quick data movement is a fundamental requirement of the kernel and its abstraction because applications that get data from peripherals will run on them. Within the benchmarking scope in section 1.4, communication between the processor and its peripheral modules are one of the main tasks of applications.

3.3 Resource utilization

3.3.1 Memory usage

The memory usage is termed as the space in the memory expressed in bytes assigned to a data structure of the FreeRTOS kernel and its hardware abstraction or an application. As mentioned in section 2.1.3, the memory management is implemented by maintaining a heap structure. The RTOS kernel has to allocate RAM each time a task, queue or semaphore is created. The *malloc()* and *free()* functions are be used for the memory allocation. The memory usage of a given data structure is the amount of heap used in the *malloc()* function when the given data structure is created.

3.3.2 Program memory usage

The usage of program memory is termed as the space in the memory consumed to store the program code after compilation. Normally it consists of application code and the code of the FreeRTOS kernel and its hardware abstraction. As the benchmark scope describes in section 1.4, the measurement of program memory usage only benchmarks the memory usage of the FreeRTOS kernel and its hardware abstraction.

Chapter 4

Benchmark of FreeRTOS and its hardware abstraction

Issues related to the benchmark of FreeRTOS and hardware abstraction are illustrated in this chapter. Requirements, techniques and benchmark attributes are discussed, which are the fundamental knowledge for the development of the benchmark tool.

4.1 Requirements

When techniques are developed and adapted to a given hardware platform, the following requirements should be followed:

- The measurements should be as accurate as possible. As mentioned in section 4.3.1, in this thesis work, a timer of a given processor is used to read the clock of the processor and to make timestamps. It is used as the time stamp in the benchmark tool, which has a granularity of microseconds.
- The solutions developed should be as non-intrusive as possible. The measurement functions should not interfere with other applications or parts of the system. For example, actions like overwriting the memory space of other applications or the kernel should not be performed.

- The measurement should give average results. This is to ensure that if the same measurements are repeated the same values should be gained as time stamped before. For any set of values, taking a large number of measurements, over a sufficiently large period of time, is the way to achieve this. For example, in this thesis work, each metric is measured typically a hundred of times. This is to ensure that instantaneous effects (like a kernel initialization) will be averaged out.
- The measurements should illustrate the value distribution for a given metric. The distribution is caused by instantaneous effects, the ramp-up time and branch executions, which represents the jitter of the execution of a function. The distribution is gained by time stamping both the maximum and minimum values in a benchmark exercise and calculating their variations.

4.2 Assumptions

In a benchmark exercise, restrictions have to be made in order to get analyzable programs. These restrictions consist of:

- Peripheral and CPU resources are assumed to be available.
- Communications between the processor and its external devices are valid, e.g. acknowledgements and data between the processor and its peripheral devices are available according to protocols.
- Tasks are assumed to never block on events or resources.

4.3 Measurement tools

4.3.1 On-chip timer based software tool

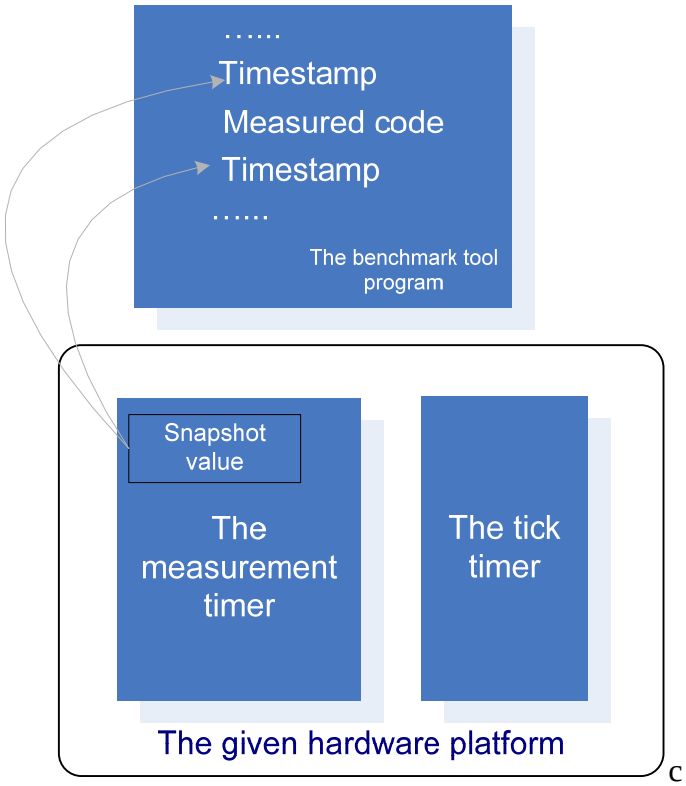


Figure 4 On-chip timer based software tool

To ease benchmark exercises, a software measurement tool is required in the benchmark tool. The software measurement tool measures metrics by using the facilities on the hardware platform. Inside the benchmark tool, a measurement timer, besides the tick timer for the FreeRTOS kernel, is required for the timing measurement. The measurement timer captures time points with timestamps, as shown in Figure 4. Using such a timer has more advantages than using a tick timer. The counter of the tick increases when a tick interrupt is issued. If the program enters the segment of code with interrupts disabled, tick interrupts will be delayed. This results a miscounting of the tick and leads to a major error in measurements. Choosing a secondary timer solves this problem. The secondary timer is independent

of the tick scheduling and does not need interrupts to increase its value. Another advantage is that a measurement tool with the independent measurement timer is able to give timestamps before the system scheduler begins. These two factors give a measurement tool with the measurement timer a larger range in measurement.

4.3.2 Oscilloscope based hardware tool

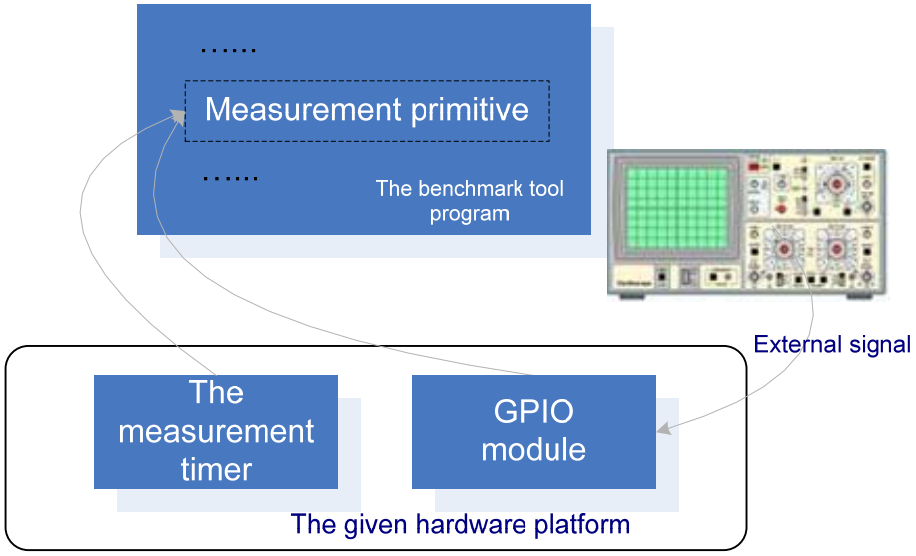


Figure 5 Oscilloscope based hardware tool

An oscilloscope based hardware tool is used to examine the accuracy of the software measurement tool, shown in Figure 5. The oscilloscope has the nature of both the higher resolution and reliability, which is suitable for identifying the accuracy of the software measurement tool. The software measurement programs generate external signals (e.g. GPIO signals) when timestamps are created and these signals are captured by the oscilloscope. The timing measurements are shown in the oscilloscope and compared to the timing measurements from the software measurement tool.

Errors of the oscilloscope based hardware tool are calculated: One part of the error is generated by the resolution of the oscilloscope which is 3.3×10^{-3} µs and the other part is created by the latency of generating an external signal by the processor, which is specific for a given processor.

4.4 Measurement techniques

In section 1.6, a number of tools and methodologies in the related work are listed. With the system utilities available and the measurement tool designed in section 4.3.1, the method of system time stamping is chosen. For the benchmark of program memory usage, due to a wide range of compilers for different hardware platforms, another specific method is used. In the following sections, the proposed methods of the benchmark are detailed.

4.4.1 Latency

The latencies of functions can be examined by measuring the execution time. The on-chip timer based software tool gives timestamps at both the beginning and the end of a given function. Their timing difference denotes the duration of executing a given function, which is the latency of the given function. Two methods of latency measurement are concerned. One is to execute a given function with a number of iterations. The latency is gained by dividing the total execution time by the number of iterations. This method gets the average latency without the value distribution. Because of the value distribution requirement in section 4.1, an alternative method is used. The given function is executed a number of times and each latency value is measured. The value distribution of the latency is used to identify conditions influencing the performance. The process of latency measurement is illustrated in Figure 6. The flow arrows indicate the process of the software program while the arrowed connectors indicate the flow of latency calculation.

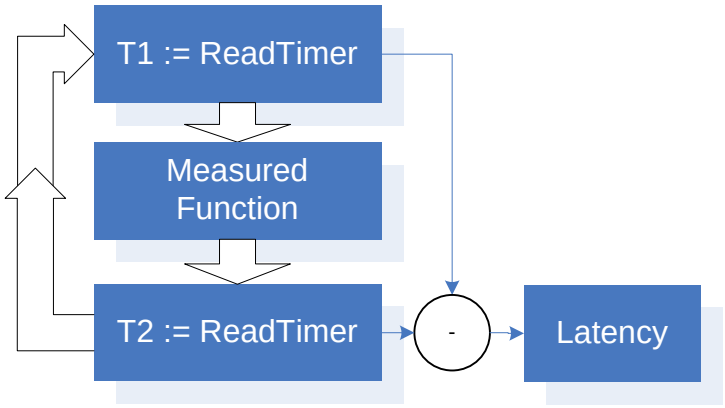


Figure 6 Measurement technique of latency

The pseudo code is shown below:

```
for(){
      T1:= ReadTimer;
      Function call
      T2:=ReadTimer;
      Latency:=T2-T1;
      }
```

To increase the accuracy, two effects are excluded. One is the influence of the context switch. The context switch is issued with a fixed period and has the possibility of taking place during a latency measurement, which will spoil the measurement with the execution time of a context switch. To exclude this overhead, the latency measurement is carried out while interrupts are disabled.

The other factor is the overhead of measurement programs. The software programs introduce the overhead of program jumps and memory write/read. To exclude these effects, the latencies of measurement programs were identified in the benchmark tool by calculating the timing difference of two successive timestamp functions. The method of identifying the latency of measurement programs is the same as shown in Figure 6. The benchmark results are shown excluding this overhead. After these two effects are excluded, the process of latency measurement is illustrated in Figure 7.

The flow arrows indicate the process of the software program while the arrowed connectors indicate the flow of latency calculation.

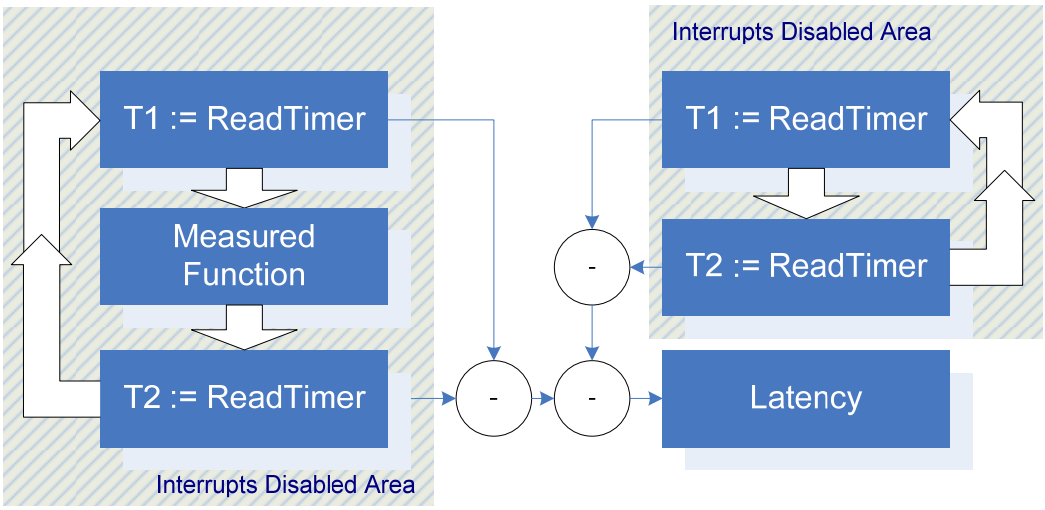


Figure 7 Measurement technique of latency (exclude overhead)

The pseudo code for latency measurement excluding the overhead is:

```
for(){
      Interrupts disabled;
      T1 := ReadTimer;
      T2 := ReadTimer;
      Interrupts enabled;
      Overhead := T2-T1;
      }
      …
for(){
      Interrupt disabled;
      T1 := ReadTimer;
      Function call
      T2 := ReadTimer;
      Latency := T2-T1-Overhead;
      }
```

Because each measurement exercise is competed within a single iteration, the measurement results exclude the overhead of the *for* loop.

4.4.2 Memory Usage

As mentioned in section 3.3.1, the memory usage of a given data structure can be measured by calculating the amount of heap used in the *malloc()* function when the given data structure is created. Because the index of the heap can not be read directly, an alternative method is used. The pointer in the C language is used as a mark to stamp the address of the heap. Because the heap structure is a continuous space of memory, the difference between the two marks is the amount of heap used, denoted by the address unit. After dividing by the size of the data type of the heap, the result is converted to heap usage, denoted by byte. The address of the heap can be accessed by calling the memory function *malloc()*. The process of memory usage measurement is illustrated in Figure 8.The flow arrows indicate the process of the software program while the arrowed connectors indicate the flow of latency calculation.

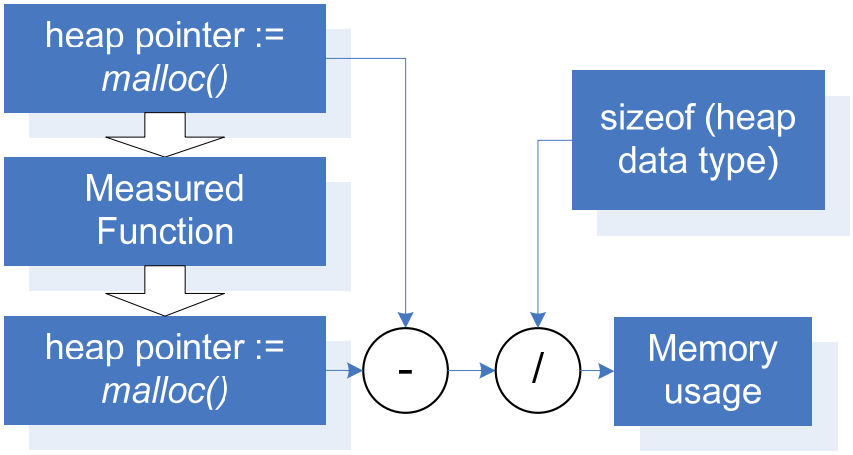


Figure 8 Measurement technique of memory usage

The pseudo code of measuring the memory usage is:

```
      Beginning heap pointer := malloc();
      Function call
      Ending heap pointer := malloc();
      Memory usage := (Ending heap pointer - Beginning heap
pointer) / sizeof (heap data type);
```

Using the memory allocation function to get the heap pointer has overhead. Every call to the memory allocation function uses at least one byte of the heap. This overhead is excluded in the results of memory usage measurement. The process of memory usage measurement is illustrated in Figure 9.The flow arrows indicate the process of the software program while the arrowed connectors indicate the flow of latency calculation.

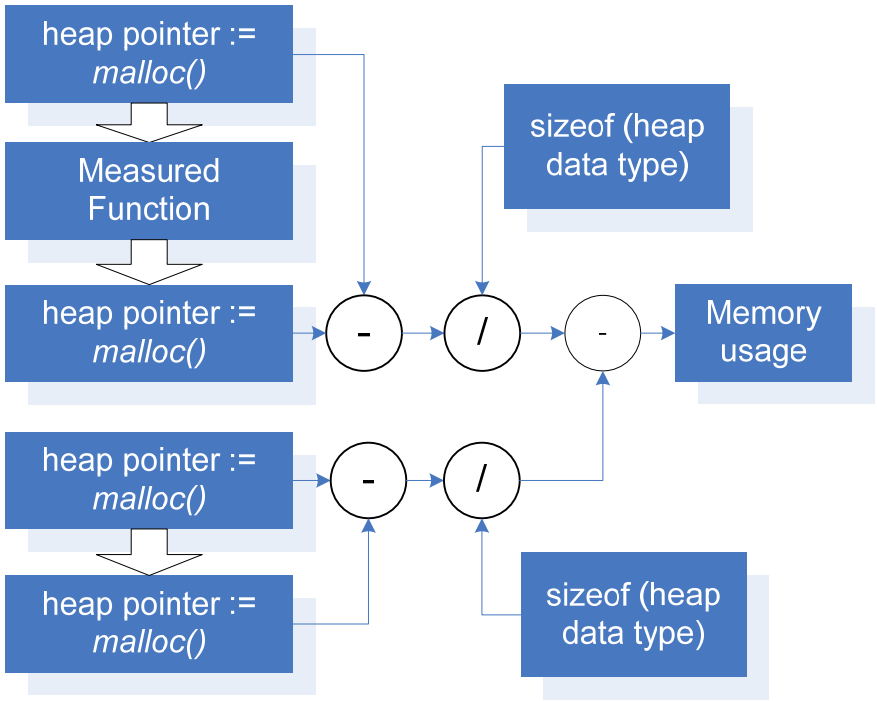


Figure 9 Measurement technique of memory usage (exclude overhead)

```
      Beginning heap pointer := malloc();
      Ending heap pointer := malloc();
      Overhead := (Ending heap pointer - Beginning heap pointer) /
sizeof (heap data type);
      …
      Beginning heap pointer := malloc();
      Function call
      Ending heap pointer := malloc();
      Memory usage := (Ending heap pointer - Beginning heap
pointer) / sizeof (heap data type) -Overhead;
```

The pseudo code of measuring the memory usage exclusive of the overhead is illustrated above:

A subset of functions with memory usage can not be measured with sequential measurement functions, like the scheduler function. To be non-intrusive to the FreeRTOS kernel and its hardware abstraction code, the measurement functions pair for scheduler memory usage is placed close to the beginning of the scheduler function and the ending point of the execution of the scheduler function.

4.4.3 Program code size

The program code size is shown by a *map* file created by the compiler. During the compilation time, functions and static variables are assigned to a segment of the program memory, whose allocations are indicated by the *map* file. By examining the *map* file, the usage of program memory of each function can be calculated.

Due to various compilers, the appearance of the map file varies. Most of the map files are created by adding a *map* flag in the compiler programs.

4.4.4 Adaption to the benchmark tool

In the stage of the benchmark tool development, the benchmark tool was adapted to the new conditions found influencing the performance. Because results from the benchmarking exercise should be reproducible and consistent, required in section 4.1, a large value distribution in the results of one metric may indicate a new condition influencing the performance. Investigation was carried out during the development stage of the benchmark tool. The same metrics under different conditions were sorted and the metrics were benchmarked separately by conditions. If the output results agree with the consistency requirement, the identification work is finished. Otherwise, a further separation work was carried out or a new way of separation was proposed. A maximum deviation rate is set. Deviations below this rate indicate consistent results.

The process of re-designing the benchmark tool to identify a given metric is illustrated in Figure 10.

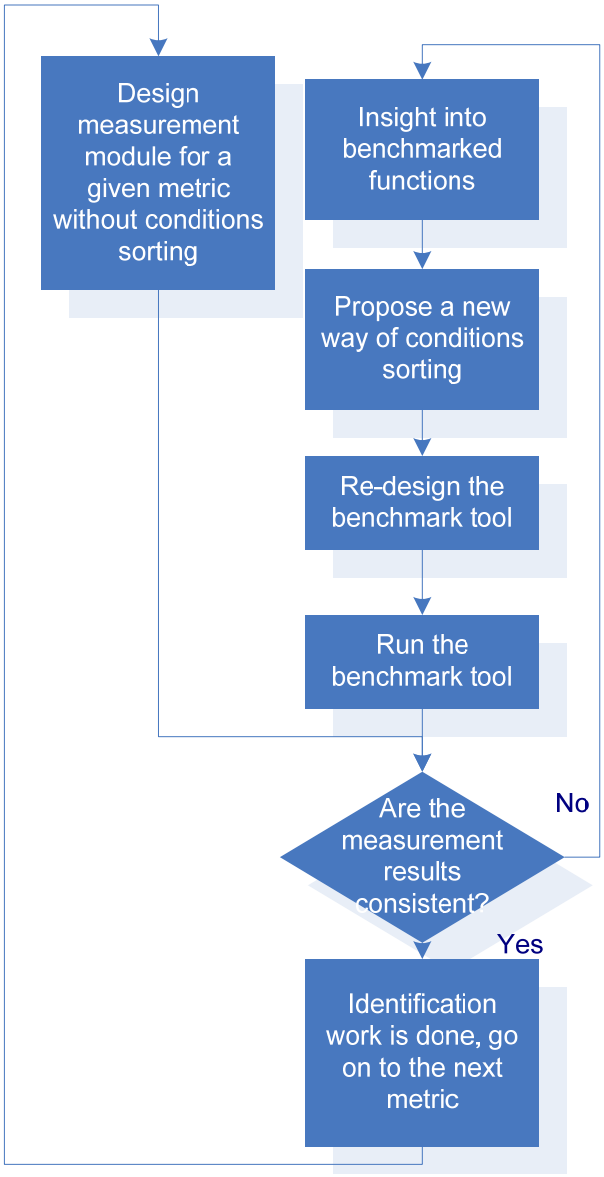


Figure 10 The process of re-designing the benchmark tool for identification

4.5 Benchmark Attributes

In Chapter 3, various metrics important for the benchmark are listed within the benchmarking scope. The values of these metrics vary with the selected hardware, and the tailoring the FreeRTOS kernel and its hardware abstraction. When

development is carried out in a wide range of functional fields and scenarios, different customizations of the software and hardware platform lead to different behaviors of the system.

This section discusses the various important factors, which cause variation in the observed performance. These factors consist of the attributes of the hardware and the favors of the kernel, which are listed in Figure 11.

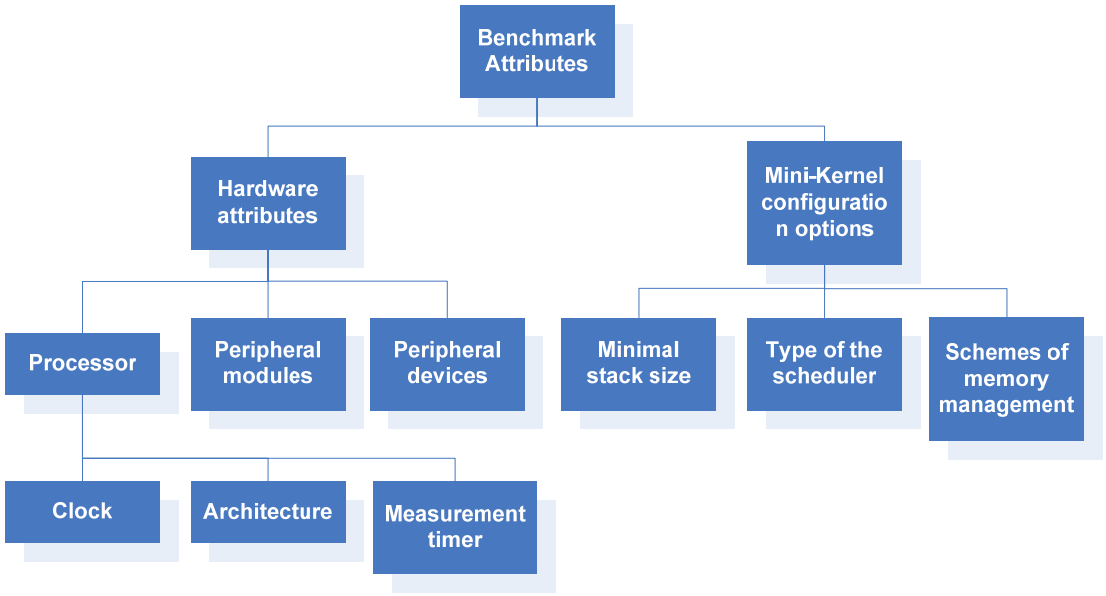


Figure 11 Benchmark attributes

A set of attributes is predefined in the tool when the benchmark tool is executed on a given sensor node. The attributes are selected according to the most common usage by projects. These parameters are also illustrated in this section, which establish the configurational environment of the benchmark.

4.5.1 Hardware attributes

The attributes of the hardware consists of the processor, its peripheral modules and peripheral devices attached.

The clock and the architecture of the processor are important parameters that affect the performance of FreeRTOS and its hardware abstraction, especially the latencies of functions. When the system clock of the processor is running, the program has a sequential execution, so the latency is related to the clock frequency. The computer architecture of the processor (MIPS, RISC and CISC etc.) defines the structure of a processor and permanently affects the performance.

As discussed in section 4.3.1, the measurement timer is used to capture time points and gives timestamps. The period of this timer defines the latency measurement range of the benchmark tool, as the timestamps are represented by the snapshot values of the timer count. A latency measurement exceeding the period of the timer is ambiguous and not interpretable. The period of the measurement timer is suggested to be configured to its maximum value.

4.5.2 Mini-Kernel configuration options

FreeRTOS is designed to be portable and configurable. A number of configurable parameters exist that allow the FreeRTOS kernel to be tailored to particular applications. This section lists representative configuration options influencing characterizations of FreeRTOS.

The minimal stack size of the FreeRTOS kernel is defined by users. When a scheduler function runs, it creates an idle task with the scheduler. The value of the minimal stack size is used to define the depth of the idle task. Thus, the value of minimal stack size influences the amount of memory usage of the scheduler.

As mentioned in section 2.1.1, FreeRTOS scheduler can be configured as preemptive or collaborative. The real time behavior of the system requires preemptive scheduling. For simpler systems, collaborative scheduling can be used. In the collaborative scheduling, a ready task can not be preempted and runs until it releases the CPU. In

the benchmarking scope of section 1.4, the preemptive scheduling is used in the benchmark tool.

As mentioned in section 2.1.3, FreeRTOS provides three different schemes for memory management. In the benchmarking scope (section 1.4), a set of applications is loaded with periodic tasks and has restrictions on power consumption. The first memory management scheme was chosen to achieve a low power consumption and real time behavior in the benchmark tool.

Chapter 5

The Benchmark Tool

This section discusses the issues of the benchmark tool. The hardware requirements for benchmark are introduced first. Design issues in the development stage are discussed, such as the selection of primitives to be benchmarked, testing applications design in the benchmark tool and the process of identification of conditions and adaption. Then the benchmark tool is introduced. The system architecture, configurable architecture and software insight of the benchmark tool are described. At last, the software monitoring facilities in the benchmark tool is introduced.

5.1 Hardware requirements for benchmark

As discussed in section 4.3, the measurement tools are involved with a set of hardware. The benchmark tool has a set of hardware requirements for the hardware platforms. First, the hardware requirements for the benchmark tool contain the minimum hardware requirements of the FreeRTOS. These include a processing unit, memory and a tick timer. Secondly, the hardware requirements also include the minimum hardware requirements of the measurement tools, as discussed in section 4.3, which include a measurement timer, a UART peripheral module, a GPIO peripheral module and an oscilloscope (as discussed in section 4.3). At last, additional hardware is required in accordance with the benchmarking specifications (discussed in section 5.3.4).

To benchmark a subset of peripheral functions of the hardware abstraction, their corresponding peripheral modules (GPIO, UART, SPI or I2C modules) on the

microcontroller and some external peripheral devices are required. An I2C external device is required if the I2C peripheral module is benchmarked. The I2C external device generates acknowledgements in accordance with the I2C protocol, which is required by the benchmark tool.

Accessorial hardware is required in benchmark exercises. To identify the error of the ported software measurement tool, as discussed in 4.3.2, an oscilloscope is required. After the metrics are measured by the benchmark tool, one UART port on the microcontroller is selected to output the benchmarking data to a desktop computer. The results are displayed in the HyperTerminal program (TechNet 2005) on the desktop computer. This proposed method requires that at least one UART peripheral should be available as a peripheral module of the hardware, a desktop computer.

5.2 Benchmark tool design

5.2.1 Benchmarked Primitives

In accordance with the metrics for benchmark listed in Chapter 3, a number of primitives are selected in the benchmark tool. These primitives are introduced in this section.

Task delay is a common system utility. It is used to delay a task for a given number of ticks. It is useful in synchronization and task control. Programmers dimension the delay time units with a number of ticks.

Task create function creates a new task and adds it to the list of tasks that are ready to run.

Task Enter/Exit critical functions pair marks the start of a critical code region, where preemptive context switches cannot occur. The benchmarking program measures the functions pair without any code in between.

A queue is a particular kind of component in which the entities are kept in order. The principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. *Queue create* function creates a new queue instance. It allocates the storage required by the new queue and returns a handle to the queue. *Queue send* and *Queue receive* functions are used to send an item to and get an item from the specific queue.

A semaphore, in computer science, is a protected variable (an entity storing a value) or abstract data type (an entity grouping several variables that may or may not be numerical) which constitutes the classic method for restricting access to shared resources, such as shared memory, in a multiprogramming environment.(Wikipedia 2008) *Semaphore create* function creates a semaphore by using the existing queue mechanism. The queue length is one as this is a binary semaphore. The data is a null type as no data is stored. *Semaphore Give* and *Take* functions are used to release and obtain a semaphore.

General Purpose Input/Output (GPIO) can act as input, to read digital signals from other parts of a circuit, or output, to control or signal to other devices. *GPIO creation* function reserves and configures the GPIO lines. *xGpioPeripheralReceive* and *vGpioPeripheralSend* set the GPIO lines used in the *PeripheralHandle*.

A Universal Asynchronous Receiver/Transmitter (UART) is a type of "asynchronous receiver/transmitter", a piece of computer hardware that translates data between parallel and serial forms. UARTs are commonly used in conjunction with other communication standards such as EIA RS-232. A UART is usually an individual (or part of an) integrated circuit used for serial communications over a computer or peripheral device serial port. UARTs are now commonly included in microcontrollers.

A dual UART or DUART combines two UARTs into a single chip. Many modern ICs now come with a UART that can also communicate synchronously; these devices are called USARTs. *xUartPeripheralInit* reserves and configures the UART interface. *xUartPeripheralSend* and *xUartPeripheralReceive* sends and receives a byte through the UART interface.

The Serial Peripheral Interface Bus or SPI bus is a synchronous serial data link standard named by Motorola that operates in full duplex mode. Devices communicate in master/slave mode where the master device initiates the data frame. Multiple slave devices are allowed with individual slave select (chip select) lines. Sometimes SPI is called a "four wire" serial bus, contrasting with three, two, and one wire serial buses. *xSpiPeripheralInit* configures the SPI interface and create driver structure. *xSpiPeripheralTransfer* makes a transfer through the SPI interface . *xSpiPeripheralReceive* receives one byte of data from the receive queue.

I²C (Inter-Integrated Circuit) is a multi-master serial computer bus invented by NXP that is used to attach low-speed peripherals to a motherboard, embedded system, or cell phone. *xI2cPeripheralInit* reserves and configures the I2C interface. *xI2cInitQueueReceive* initiates receiving bytes from the I2C interface into the receiving queue. *xI2cPeripheralReceive* receives a byte from the I2C receiving queue. *xI2cPeripheralSend* sends a byte through the I2C interface.

5.2.2 Testing applications in the benchmark tool

A number of testing applications are used in the benchmark tool for benchmark the hardware abstraction. Testing applications were designed to match the communication protocols of the external devices. Testing applications were also designed to be less hardware-demanding and non-influencing to the benchmarking results. Special design issues are discussed in this section.

In the measurement of the UART, SPI and I2C data receiving functions, items are supposed to be received from external devices. To ease the testing applications, these items are created by the software. The testing applications do not influence the latency measurements, as in the peripheral data receiving functions is the time to fetch an item from the peripheral data receiving queue. The process of interrupt service routines is excluded. An item is created by the benchmark tool and pushed into the peripheral queuing system in advance. The latency of data receiving function is measured using the technique in section 4.4.1 on the process of fetching the item from the peripheral queuing system.

In the measurement of the UART, SPI and I2C data sending functions, the item to be sent is selected to be non-intrusive. The item to be sent by the UART peripheral function is a NULL item, which is not shown in the results. The item to be sent by the SPI peripheral function is a NULL command, which does not affect the status of the SPI external device. In the I2C part, to successfully receive the acknowledgement from the I2C external device, the item to be sent is the address value of the I2C external device. According to the I2C protocol, such an item targets at the selected I2C external device and acknowledged by the I2C device.

5.2.3 The process of identification of conditions and adaption

As discussed in section 1.2, one of the objectives is to identify conditions influencing the performance of the FreeRTOS kernel and its hardware abstraction by using the benchmark tool. The identification work was carried out by adaption to the benchmark tool in the development stage. The techniques of adaption are discussed in section 4.4.4. The maximum value of results distribution is set to 10%. The rate of deviation exceeding this value indicates that one or more non-identified conditions exist. Further sorting work was carried out to investigate conditions.

A number of measurements were carried out to get the results distribution. A subset of functions is executed a hundred times in a benchmark exercise, which are termed

as multiple-executable functions. For other functions, termed as single-executable functions, the number of executions within a benchmark exercise is limited by the resource of the hardware platform. Some executions of functions are limited by the amount of memory due to the memory management theme selection in section 2.1.3. The memory space does not allow a memory consuming function to be executed one hundred times. Others executions are limited by the peripheral resources. Each peripheral module is allowed to be registered only once. For single-executable functions, multiple benchmark exercises are required to get the value distribution. These two ways of identification can be illustrated in Figure 12.

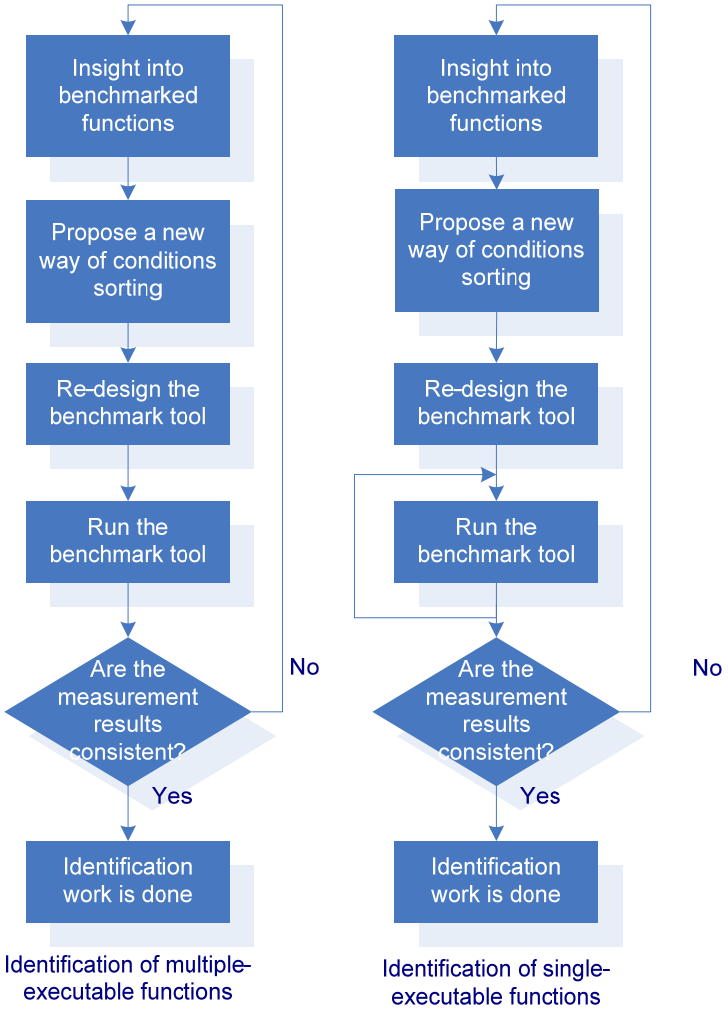


Figure 12 The process of identification of conditions

5.3 Insight of the benchmark tool

5.3.1 System architecture of benchmark tool

The benchmark tool is designed as an application running on the FreeRTOS kernel and its hardware abstraction. As stated in section 4.1, the benchmark tool is designed to be non-intrusive to the FreeRTOS kernel and its hardware abstraction. The benchmark does not make changes to the source code of the FreeRTOS kernel and the peripheral drivers. It does not add facilities to the FreeRTOS kernel and its hardware abstraction and it runs in the application layer of the system. The system structure of the benchmark, the hardware and the FreeRTOS kernel and its hardware abstraction is illustrated in Figure 13.

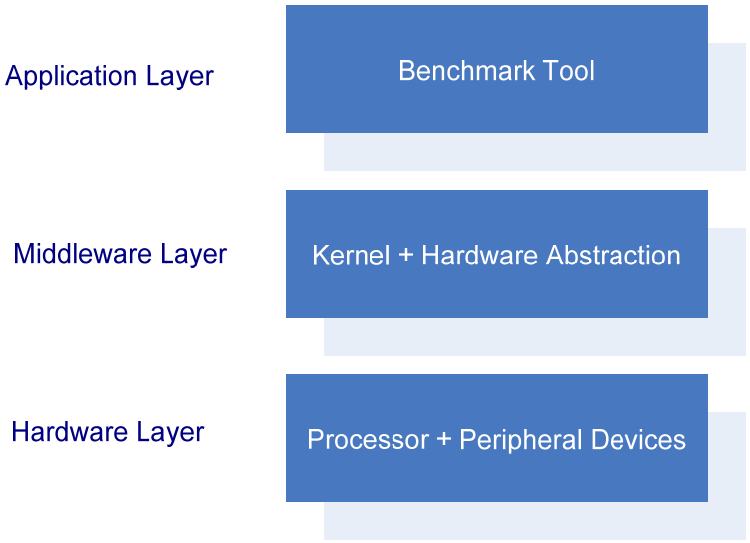


Figure 13 System Architecture of Benchmark

The lowest layer is the hardware layer. As the hardware requirements in section 5.1, this layer contains peripheral devices and a processor with peripheral modules. The layer in the middle is the FreeRTOS kernel and its hardware abstraction, including the FreeRTOS kernel and the hardware abstraction, illustrated in Figure 2. The top layer

is the application layer. During a benchmarking period, the benchmark tool is the only application running in it.

5.3.2 Configurable architecture of the benchmark tool

The benchmark is designed to be configurable. With the presentation of a port of the given hardware platform, the benchmark tool is used to benchmark a new hardware platform after configuration. Both the FreeRTOS kernel and its hardware abstraction and the benchmark tool consist of common parts and configurable parts. For a given hardware, the software architecture for the benchmark process is illustrated in Figure 14.

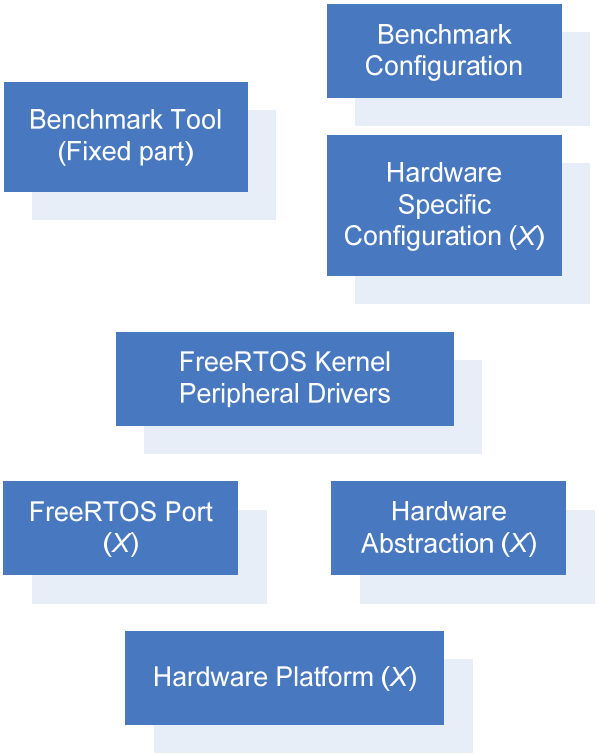


Figure 14 Software Architecture for a given hardware (X denotes a given hardware)

The FreeRTOS kernel and its peripheral drivers are consistent, with interfaces to hardware. The ports for FreeRTOS and its hardware abstraction are specific to

hardware. On top of the FreeRTOS kernel and its hardware abstraction, the benchmark tool is present. It consists of a fixed part and two configurable parts.

The fixed part is the main part of the benchmark tool. In the stage of identifying conditions, metrics were investigated with individual condition as discussed in section 5.2.3. The code of the benchmark tool was adapted to examine these conditions. The conditions, which are not dimensioned by parameters, were adapted to the benchmark tool. For the parameter dimensioned conditions, users can investigate these conditions by changing parameters in the attributes configuration part. The hardware specific configuration part of the benchmark tool is adapted to the hardware specific structure of a given hardware platform. The detailed configuration method is illustrated in section 5.3.4.

5.3.3 Software insight of the benchmark tool

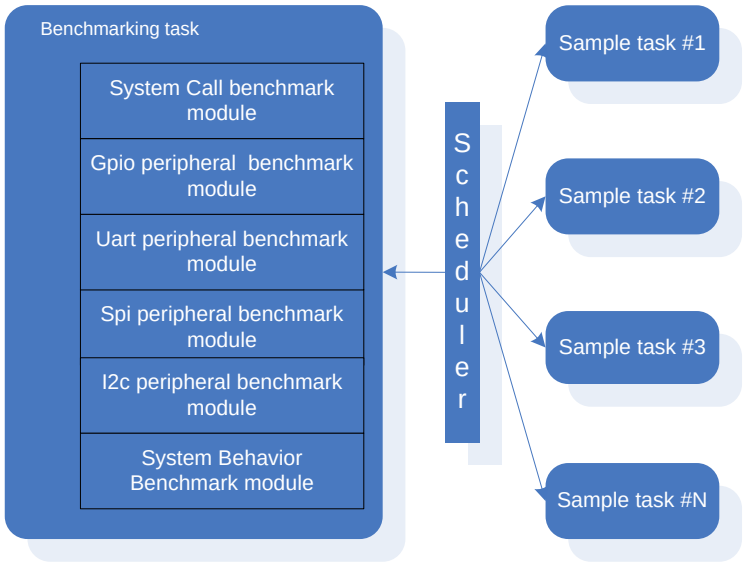


Figure 15 Run-time multi-task structure

The source files of the benchmark tool consist of three files: *benchmarkMain.h*, *benchmarkMain.c* and *measureTool.c*. *benchmarkMain.h* is the configuration file, which consists of a benchmark configuration part and a hardware specific part.

benchmarkMain.c is the fixed part of the benchmark tool in Figure 14. *measureTool.c* is used to port the measurement tool to the benchmark tool.

When a benchmark exercise is carried out, the run-time multi-task structure is as illustrated in Figure 15. The benchmark application creates a benchmark task and a number of sample tasks. The main benchmark work is assigned to the benchmark task. According to the metrics for benchmark in Chapter 3, the benchmark tool has six benchmark modules: the system call benchmark module, the GPIO peripheral benchmark module, the UART peripheral benchmark module, the SPI peripheral benchmark module, the I2C peripheral benchmark module and the system behavior benchmark module. The sample tasks are created to benchmark the system calls and investigate the multi-task feature of the FreeRTOS kernel. The sample tasks are also used as indication of the status of the system in the run-time. This feature is discussed in section 5.4.

5.3.4 The use of the benchmark tool

In this section, the process of using the benchmark tool is introduced. After a given hardware is selected, there are five steps to utilize the benchmark tool, which are illustrated in Figure 16.

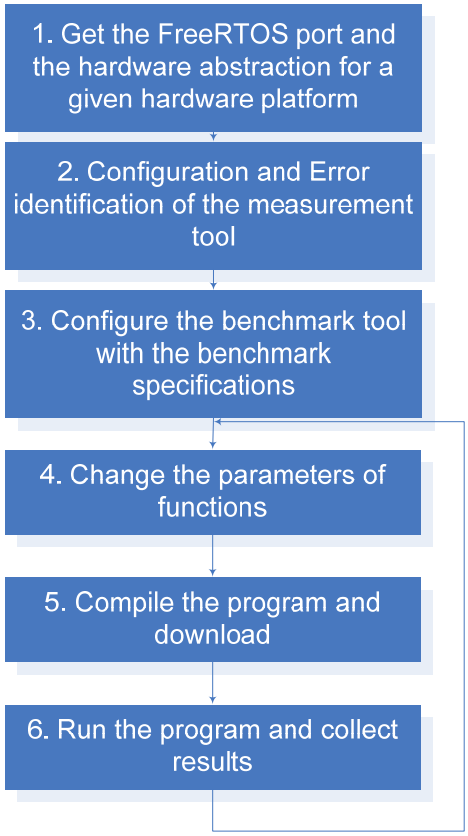


Figure 16 The process of using the benchmark tool

In the first step, as the pre-requirement of a benchmark exercise, a FreeRTOS port with the hardware abstraction is required. Software development is carried out in advance to port the FreeRTOS kernel and its hardware abstraction onto the hardware. Before the benchmark stage, both the software and the hardware platform should be available.

In the second step, the software measurement tool, discussed in section 4.3.1, is configured to the hardware. This step consists of the following work:

- Configure the measurement timer in the *measureTools.c* file and *benchmarkMain.h* file. The measurement timer for benchmarking is configured using the timer peripheral functions with the *Interrupt & Continue & Capture* mode(Philips 2006; TexasInstruments 2006). The period (Philips 2006; TexasInstruments 2006) of the timer is set to its maximum value. The

values of the period and the divider of the measurement timer are declared by re-defining the string identifiers in *benchmarkMain.h* file.

- Select the indexes of the UART and GPIO peripheral modules in accordance with the hardware connection by re-defining string identifiers in *benchmarkMain.h* file. As required in section 5.1, one of the UART peripherals and one of the GPIO peripherals are selected in accordance with the hardware connection.

- Indicate the clock frequency. The value of the clock frequency is re-defined as a string identifier in *benchmarkMain.h* file.

- Set the value for the stack size of the benchmark task. Because of the large amount of measurement work loaded in the benchmark task, the stack size of the benchmark task is set large enough to keep the benchmark task running. Acquiring the value in advance is not possible, but it can be got in the debugging stage using the monitoring facilities in section 5.4.

- Evaluate the resolution of the software measurement tool. An external signal is selected and configured to be output in the measurement tool. Then the error of the measurement tool is identified by using the oscilloscope based hardware tool, discussed in section 4.3.2. The duration time of a test function is measured both by the measurement tool and the oscilloscope. The difference in results illustrates the error of the measurement tool.

In the third step, benchmark specifications are made. The benchmark specifications include three parts. The first part is to choose modules for the benchmark. String identifiers in the *benchmarkMain.h* (Demonstrated in Appendix D) file are re-defined. The second part is defining the hardware attributes of the benchmark tool, such as the data type of results, the address of the external peripheral devices, the GPIO, UART, SPI and I2C modules if selected in the first part. The third part is defining the

measurement attributes of the benchmark tool, such as the software resolution, number of iterations and data type.

The following three steps are iterative exercises of benchmarking. The fourth step changes the parameters of functions. The fifth step compiles the program and downloads it. The process of compiling and downloading is the same as a user application. The last step is to run the program and collect the results. The benchmark results are outputted to the serial port of the host computer, where the HyperTerminal program is used to receive the data and to capture them into a text file for offline investigation.

5.4 Software monitoring facilities in the benchmark tool

The benchmark tool benchmarks the commonly used system calls and peripheral functions. It is an application with heavy workload. It consumes quite a lot of the resources of the hardware. Special care should be taken to avoid the resources to be overused. Tracing facilities are provided in run-time to ease the porting work of the benchmark tool. The benchmark tool has two facilities for examination: a monitoring task and an indicator of heap usage.

The monitoring task is used to show the stability of the system. It is a LED blinking task beside the benchmark task. The blinking LED indicates that the system is stable and the program is still running. This facility is useful to detect stack overflow. As stated in (Barry 2008), stack overflow is the most common source of support requests and is not easy to detect. The monitoring task gives an intuitive way to show the application instability. If the benchmark task stops while the monitoring task, with a higher priority in the debug stage, is still running, this may be the result of a failure of creation of the benchmark task or a placement of a busy loop within the benchmark

task. If both the benchmark task and the monitoring task with higher priority stop, the program may have encountered an overflow.

The indicator of heap usage is used to show the current usage of the heap in the run-time via the serial port. A subset of functions consumes an amount of heap. For some hardware, the amount of heap defined is not enough for benchmarking all the metrics. The indicator of heap usage illustrates the current usage of heap by percentage. If some data structures are not created and the percentage of usage is high at the same time, this may indicate that heap space is not large enough for creating a new data structure.

Chapter 6

Case studies

In this chapter, the benchmarking exercises are carried out using the benchmark tool introduced in Chapter 5. Two hardware platforms were used as test beds. One is the Body Sensor Network node (BSN), a hardware platform developed by Imperial College London, designed for the ease of the development of pervasive health care system. The other one is the CoolFlux development board, a development board for Small Autonomous Network Devices (SAND). Both hardware platforms consist of a processor and peripheral hardware.

First the development environments and experimental setup are introduced in 6.1. As the first step of the benchmark, the error is identified in section 6.2. Results and the performance analysis are given in section 6.3.

6.1 Benchmarking environments

6.1.1 BSN node

Figure 17 View of a BSN node

BSN node is a hardware platform with an MSP430 microcontroller, a CC2420 radio chip and a stackable design. The BSN node provides a low-powered, miniaturised, and intelligent platform for the development of pervasive physiological and context aware sensors.

The microcontroller on the BSN node is a MSP430F149. It is a microcontroller configured with two built-in 16-bit timers, a fast 12-bit A/D converter, two universal serial synchronous/asynchronous communication interfaces (USART), and 48 I/O pins. Peripheral modules on the microcontroller consist of a UART, a SPI but without an I2C peripheral. Two GPIO lines are used to simulate the I2C protocol, controlled by software.

The source code FreeRTOS kernel is compiled by MSPGCC compiler. MSPGCC is a port of the GNU C and assembly language tool chain to the Texas Instruments MSP430 family of low-power microcontrollers(Underwood 2003). The MSPGCC port of the GNU C compiler is currently based on version 3.2.3 of GNU GCC. It supports all the current variants of the MSP430 processor, and comes with a full set

of header files for the processors. Signed and unsigned integers of 8, 16, 32, and 64 bit lengths are supported.

To meet the hardware requirements in section 5.1, a BSN board was connected to an I2C device LIS3LV02DQ, a three axis digital output linear accelerometer with an I2C serial interface. The hardware configuration for benchmarking is illustrated in Figure 18.

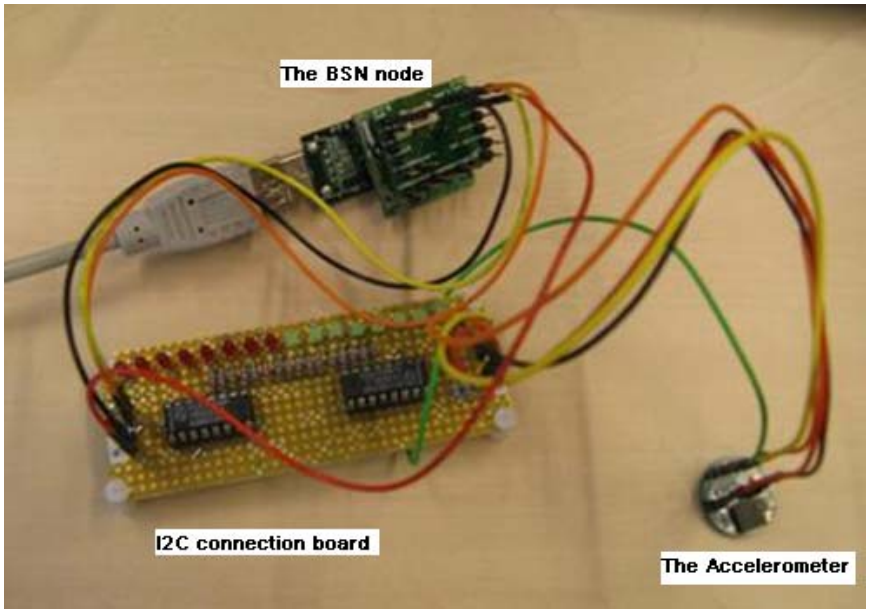


Figure 18 Hardware for benchmarking BSN board

The port for FreeRTOS on the BSN node is supported by FreeRTOS.org. Both the system clock and peripheral clocks are generated from an on-board DCO clock which has the frequency of 4.5 MHz. Timer 0 is selected as the system timer to generate ticks. The context of the BSN node includes all the general purpose MSP430 registers.

For the peripheral hardware, GPIO, UART and SPI hardware modules are present on the BSN node, but the I2C hardware is missing. A software implementation of I2C is present in the hardware abstraction. In the hardware presentation layer and the basic functions layer of Figure 2, two GPIO modules on the BSN node are used to simulate the behavior of an I2C hardware module. The levels of these GPIO lines are controlled by the processor to simulate the I2C protocol.

6.1.2 CoolFlux development board environment

The CoolFlux DSP is an ultra low-power, embedded DSP core, developed originally for audio applications in systems with ultra low power requirements. The CoolFlux DSP features a dual Harvard architecture.(Catalano 2005) It has a 24-bit data path, two 24 × 24 bit multipliers, and three ALUs and 56-bits accumulators. The CoolFlux DSP is fully interruptible, with three interrupt vectors available. Finally, the CoolFlux DSP is a fixed point processor.

The CoolFlux development board is an easy board for programming, debugging and testing. All the I/O pins are available and an oscilloscope can be plugged to those pins. A JTAG connector is also available which can be used to run the on-chip debugger and debug step by step a program on your computer. Programs are compiled by CHESS Retargetable Compiler developed by Target Compiler Technologies N.V.(N.V. 2008). A specific script is used to converts the produced *ELF* image file into an *ASCII* file, which can be downloaded into the board.

The port of FreeRTOS for the CoolFlux development board is available in Philips Research. Both the system clock and peripheral clocks are generated from an on-board DSP clock which has the frequency of 12 MHz. Timer 1 is selected as the system timer to generate ticks. The context of the CoolFlux development board includes the data path registers, the pointer registers, the stack pointer, the status register and its copy within interrupt, and finally the link register and its interrupt counterpart. (Catalano 2008) For the peripheral hardware, GPIO, UART, SPI and I2C hardware modules are present on the CoolFlux development board. The hardware configuration for benchmarking the CoolFlux development board is illustrated in Figure 19.

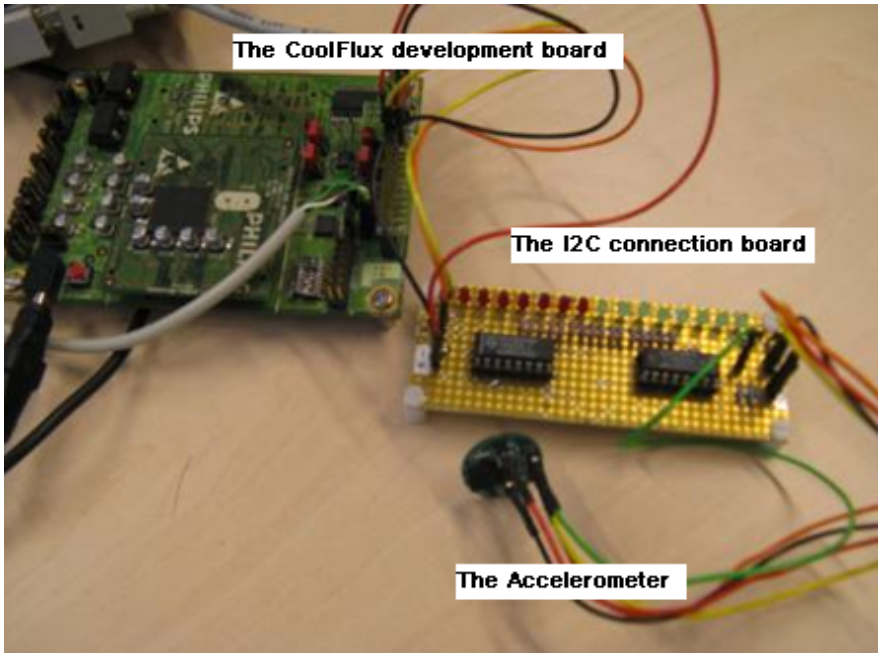


Figure 19 Hardware for benchmarking CoolFlux development board

6.2 Configuration and Error identification

As the second step of the process of the benchmark in Figure 16, individual benchmark configuration for both hardware platforms is discussed in this section and the error of the measurement tools is identified. The CoolFlux development board uses an external clock, which has the frequency of 12 MHz. The measurement timer gets the clock signal from this clock and has a granularity of 8.3×10^{-8} second. The BSN node uses the on-board DCO clock which has the frequency of 4.5 MHz. Due the timer divider of 8, the granularity of the measurement timer on the BSN node is 1.78×10^{-6} second.

The techniques of error identification are illustrated in section 5.3.4. GPIO 0 was selected as the external signal for identification for the CoolFlux development board while P5.4 port was selected as the external signal for identification for the BSN node. The error of both measurement tools are shown in Table 1 and Table 2. The error for

the measurement tool on the BSN node is 4.66% while on the CoolFlux development board it is 2.15%.

Table 1 Error of the measurement tool on the BSN node

	Average (ms)	Max (ms)	Min (ms)
Oscilloscope	0.01524	0.01528	0.01520
Software Measurement Tool	0.01453	0.01635	0.01453
Difference (%)	4.66	7.00	4.41

Table 2 Error of the measurement tool on the CoolFlux development board

	Average (ms)	Max (ms)	Min (ms)
Oscilloscope	0.00279	0.00280	0.00279
Software Measurement Tool	0.00273	0.00282	0.00265
Difference (%)	2.15	0.71	5.02

The error of the measurement tools originates from two aspects. One part of the error comes from the LED blinking functions. The LED blinking functions read and write the registers of GPIOs to generate signals for the oscilloscope. These read/write actions cause delays. The other source of errors is at the measurement functions side. The executions of the measurement functions and reading the count value of the timer take time.

6.3 Benchmarking results

A number of exercises were carried out on these two test beds. Results were collected offline by examining the text files as discussed in section 5.3.4. The individual method of benchmarking each function is illustrated first. The results of the measurements are shown in figures. Conditions are identified and performance analysis is discussed. Complete tables of results can be found in Appendix C. In this section, without any specific statement, both the single and multiple tasks environments are examined and results are illustrated with both cases.

6.3.1 Task Delay

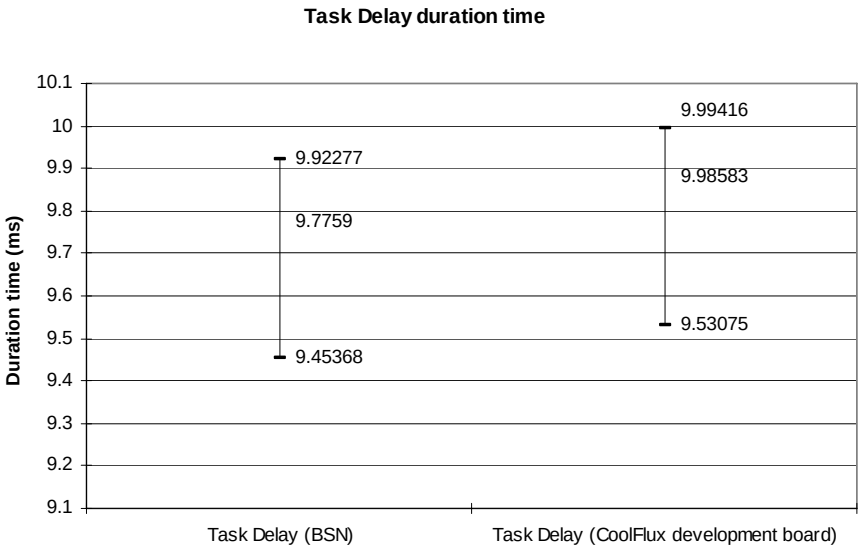


Figure 20 Task Delay duration time

Task delay function is used to delay a task for a given number of ticks. The duration time of the task delay of 10 milliseconds was measured. As illustrated in Figure 20, the output delay duration is not exact as indicated in the parameter of the function. The timing difference can not be explained as the error of the tick system, which is supposed to make the output duration around the specified delay time. Duration larger than 10 milliseconds is expected. But as illustrated in Figure 20, all the duration are below the specified duration, between 10 milliseconds and 9 milliseconds.

The scheduling mechanism of the FreeRTOS kernel accounts for this phenomenon. The scheduler checks the delayed tasks every tick period. Tasks delayed by the task delay function can only be checked at the tick point while the task delay function can be called at any place in the program. This makes the resulting delay duration shorter than the specified delay time. This effect is illustrated in Figure 21.

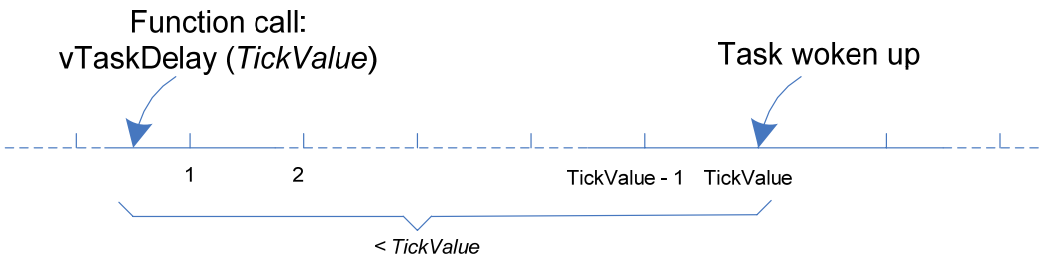


Figure 21 **Time line of Function call: *vTaskDelay(TickValue)***

6.3.2 Latencies

Latency of Task creation

For a given application in the benchmarking scope in section 1.4, a number of tasks are created. The latency of task creation was examined. As discussed in section 5.3.3, the benchmark tool creates more than one task. The duration time for each created task is measured. Because the task creation function is single-executable function, discussed in Figure 12, multiple experiments were carried out. Due to the limited memory resource on the hardware, at most three tasks are created. The latency results are illustrated in Figure 22.

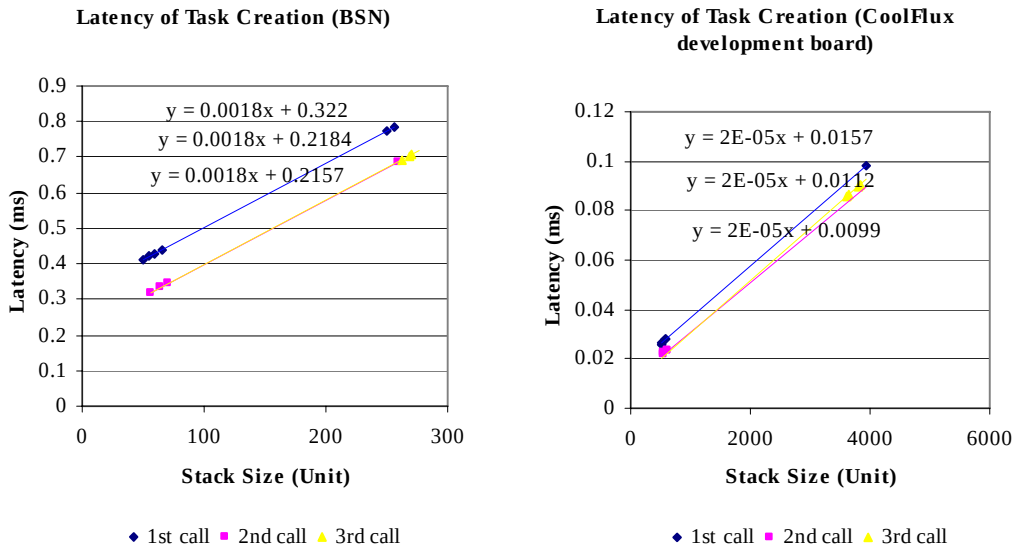


Figure 22 Latency of Task Creation

The figure shows that the latency of task creation increases as the stack size of a specific task increases. When a task is created, the system performs two memory allocations as discussed in section 2.1.4. The time for allocating the memory space of Task Control Block (TCB) is fixed. The time for initializing the task stack by filling with the fixed value increases with the depth of the task.

Figure 22 shows that the latency for task creation function depends on the number of ready tasks. The latency for a task creation gets reduced after the first task is created. Created with equal stack size, the first task creation takes longer while the following task creations take an equal amount of time. The timing difference, around 0.11 ms in the BSN node and 0.00045 ms in the CoolFlux development board, is too large to be explained as a result of the cold cache effect (Westwood 2006). Alterative explanation is used. In FreeRTOS, when the first task is created, a preliminary initialization is carried out. A function to ready all the lists used by the scheduler is called, which takes time to execute. The extra execution time causes the timing difference.

Latency of system calls

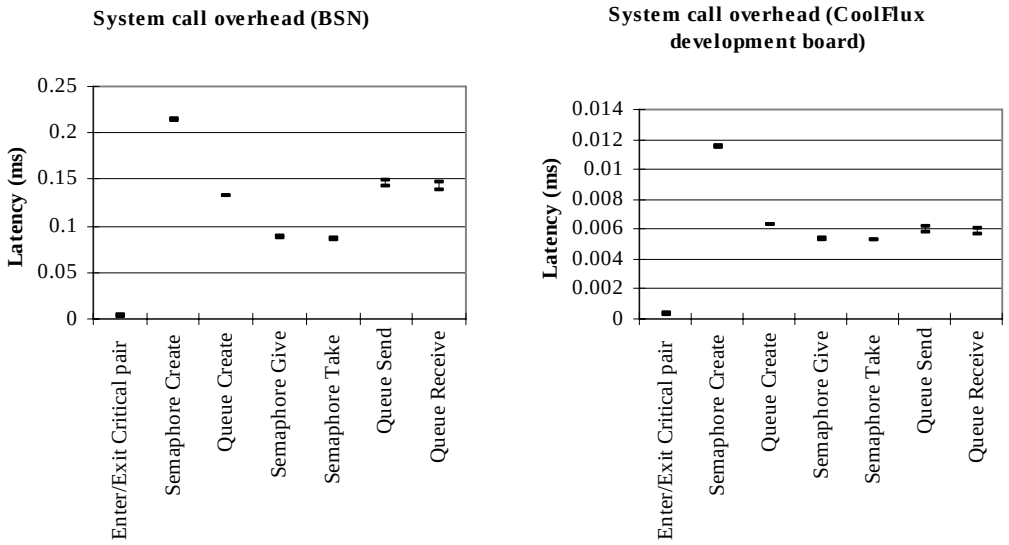


Figure 23 System call overhead

The benchmark tool measures the system call overhead in the benchmarking task, which is set with the highest priority (Figure 15). The system calls are measured sequentially. The individual system call is introduced in section 5.2.1.

Enter/Exit critical functions pair was measured without any code executed in between. This function is a multiple-executable function which is executed one hundred of time to get the value distribution as discussed in section 5.2.3.The pseudo code is shown below:

```
for   {
      T1:=The measurement function;
      Enter critical function;
      Exit critical function;
      T2:=The measurement function;
      Latency (Enter/Exit critical functions pair):=T2-T1;
      }
```

In the FreeRTOS kernel, queues are dimensioned by types: char type, short type and long type. The creation time of queues and semaphores were measured by the single-

executable method. Latencies of queue/semaphore sending/receiving functions were benchmarked by measuring the duration of sending or receiving one item into a queue or a semaphore. The duration time of queue sending and receiving are also dimensioned by three types. For individual queue/semaphore type, the multiple-executable method is used. In Figure 23, it is observed that, with different types of items, variations of their execution time are low.

Latency of GPIO peripheral functions

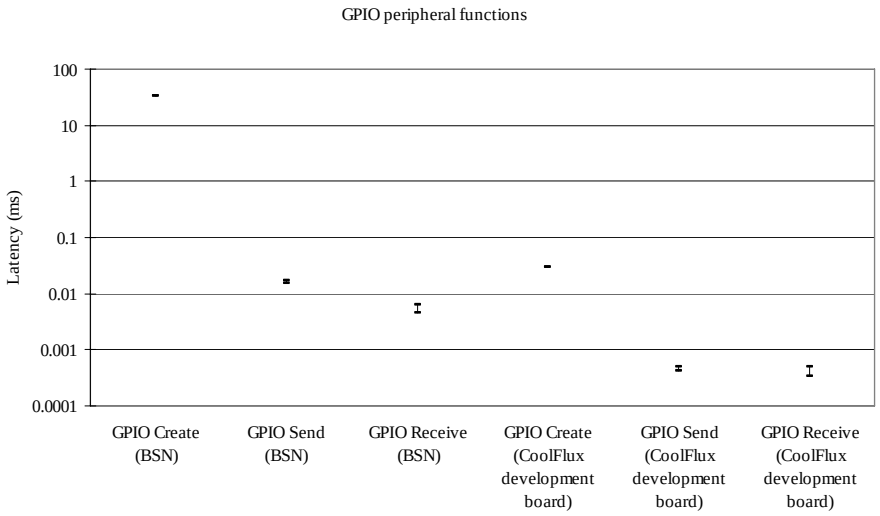


Figure 24 Latencies of GPIO peripheral functions

Because the similar functionalities of different GPIOs on the board, GPIO benchmarking is carried out by benchmarking a specific GPIO port. An accessible port by the oscilloscope is selected. GPIO 0 was selected for BSN board. GPIO 9 was selected for CoolFlux development board. The latencies of GPIO sending functions were benchmarked by measuring the duration time of setting the benchmarked port to be high level. The latencies of GPIO receiving functions were benchmarked by measuring the duration time of reading the level at a specific GPIO. The single-executable method is used in the GPIO creation function. The multiple-executable method is used in the GPIO sending and receiving functions. In Figure 24, it is

observed that, with different number of tasks, variations of their execution time are low.

Latency of UART peripheral functions

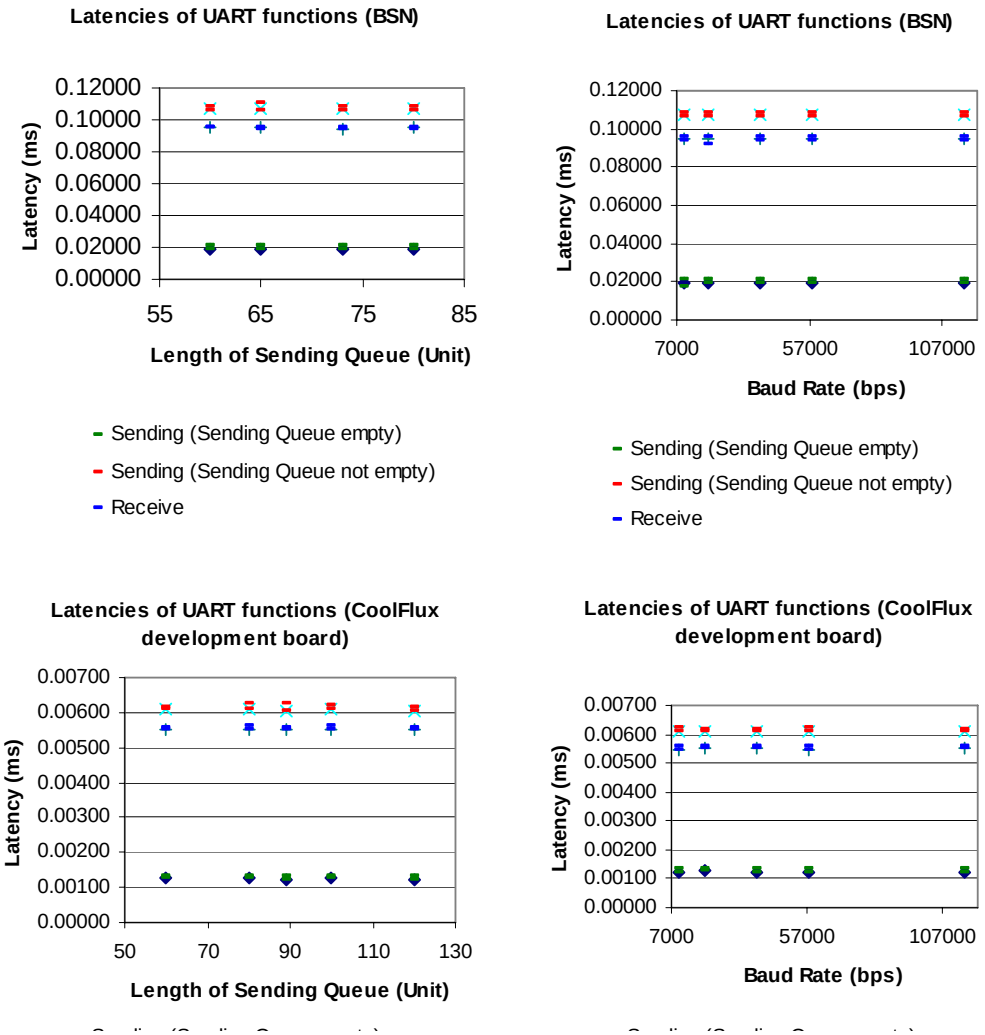


Figure 25 Latencies of UART peripheral functions

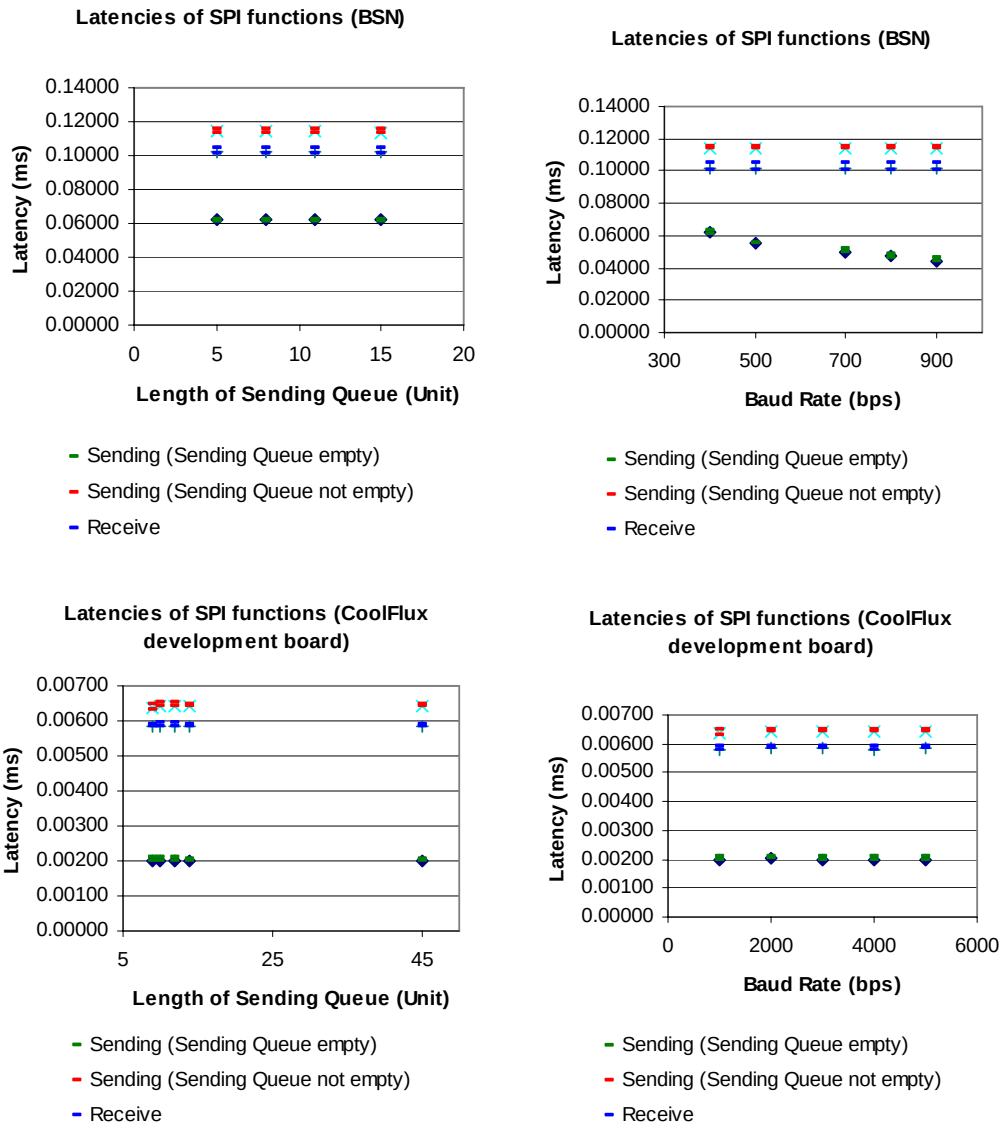
UART peripheral functions include a UART creation function, a sending function and a receiving function. The latency for the execution of a sending function is the time for the processing unit sending one byte into the peripheral memory, illustrated in Figure 3. The latency for the execution of a receiving function is the time for the

processing unit receiving one byte from the peripheral memory, illustrated in Figure 3. The single-executable measurement method is used in the UART creation function and the multiple-executable measurement method is used in sending and receiving functions. UART modules for benchmarking are selected in the favor of hardware connections. For the BSN board, UART 1 was selected. For the CoolFlux development board, UART 0 is selected.

As discussed in section 2.2.1, two conditions of the sending function are present, featured by the status of the peripheral module. Figure 25 shows that the latency of the sending function varies in these two conditions, an empty sending queue and a non-empty sending queue. It takes more time for an item sent to the peripheral sending queue than to the peripheral sending register. Another two conditions were concerned. One is the length of the sending queue. The other one is the baud rate of peripheral hardware. To investigate the relationship between the latencies and the two conditions, benchmarking exercises were carried out where one of conditions varies, while at the same time, the other one was kept constant.

From Figure 25, it is observed that the length of the sending queue does not influence the latencies of peripheral functions. The peripheral sending queue is implemented as an instance of the queue. As discussed in section 2.1.4, the sending and receiving actions of the queue are involved with the changes of the pointers to the next byte and the status values of the queue. The length of the queue does not influence the latencies of sending and receiving a byte. From Figure 25, it is also observed that the baud rate does not influence the latencies of peripheral functions. As illustrated in Figure 3, the latencies of the sending and receiving functions are the communication time between the processing unit and the peripheral memory. The baud rate of the peripheral module influences the communication between the peripheral module and the peripheral devices, but not the communication between the processing unit and the peripheral memory.

Figure 26 Latencies of SPI peripheral functions



Latency of SPI peripheral functions

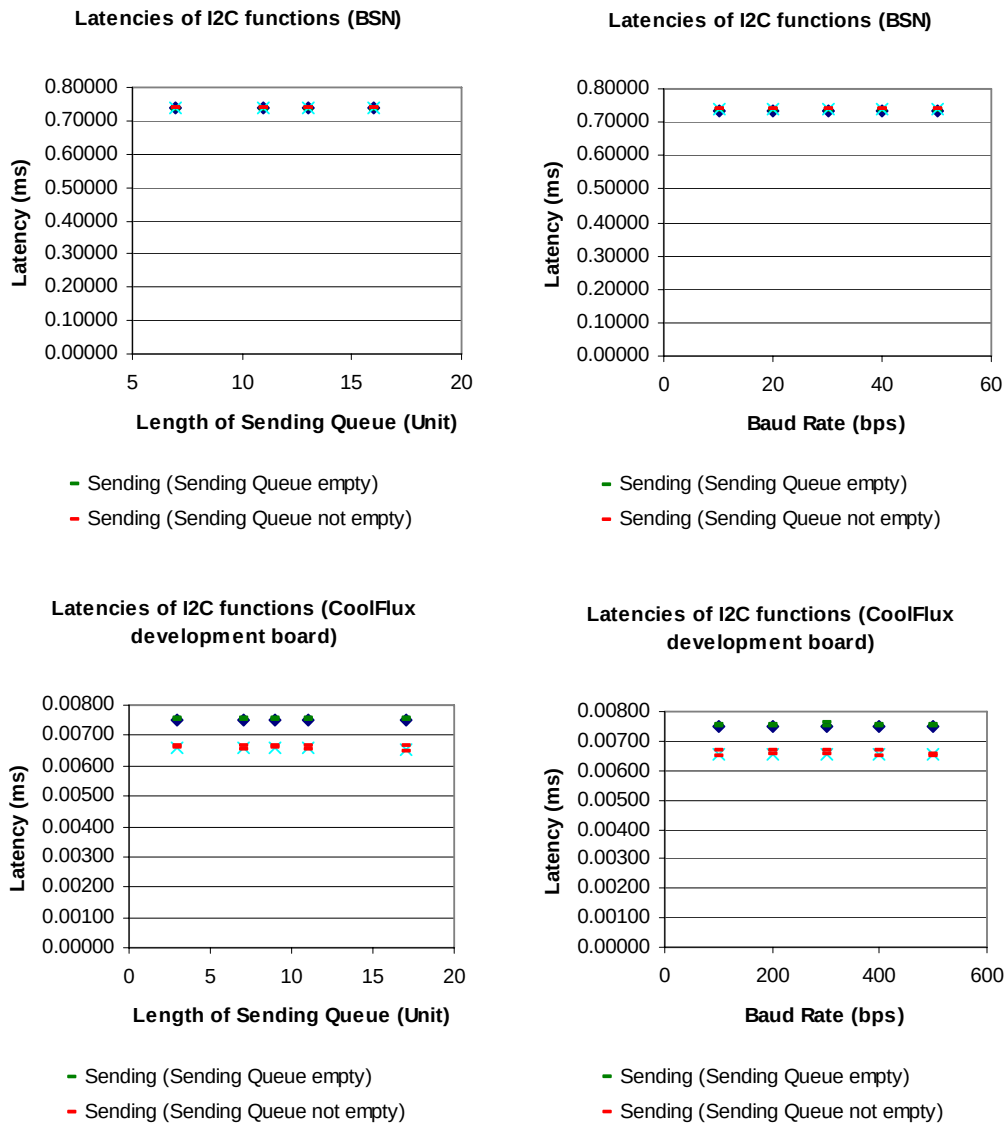
SPI peripheral functions include a SPI creation function, a sending function and a receiving function. The latency for the execution of a sending function is the time for the processing unit sending one byte into the peripheral memory, illustrated in Figure 3. The latency for the execution of a receiving function is the time for the processing unit receiving one byte from the peripheral memory, illustrated in Figure 3. The single-executable measurement method is used in the SPI creation function and the multiple-executable measurement method is used in sending and receiving functions. SPI modules for benchmarking are selected in the favor of hardware connections. For the BSN board, SPI 0 was selected. For the CoolFlux development board, SPI 1 is selected.

As discussed in section 2.2.1, two conditions of the sending function are present, featured by the status of the peripheral module, an empty sending queue and a non-empty sending queue. Figure 26 shows that the latency of the sending function varies in these two conditions. It takes more time for an item sent to the peripheral sending queue than to the peripheral sending register. Another two conditions were concerned. One is the length of the sending queue. The other one is the baud rate of peripheral hardware. To investigate the relationship between the latencies and the two conditions, benchmarking exercises were carried out where one of conditions varies, while at the same time, the other one was kept constant.

From Figure 26, it is observed that neither the length of the sending queue nor the baud rate influences the latencies of peripheral functions. The reason of this phenomenon can be referred in the UART section, as the SPI peripheral shares the same structure of hardware connection of peripherals with the UART peripheral.

Latency of I2C peripheral functions

Figure 27 Latencies of I2C peripheral functions



I2C peripheral functions include an I2C creation function and a sending function. The latency for the execution of a sending function is the time for the processing unit sending one byte into the peripheral memory, illustrated in Figure 3. The single-executable measurement method is used in the I2C creation function and the multiple-executable measurement method is used in the sending function. I2C modules for

benchmarking are selected in the favor of hardware connections. For the BSN board, I2C 0 was selected. For the CoolFlux development board, I2C 0 is selected.

As discussed in section 2.2.1, two conditions of the sending function are present, featured by the status of the peripheral module, an empty sending queue and a non-empty sending queue. Figure 26 shows that the latency of the sending function varies in these two conditions for the CoolFlux development board. It takes more time for an item sent to the peripheral sending queue than to the peripheral sending register. For the BSN node, the hardware connection of peripherals is not applicable. As described in section 6.1.2, the BSN node does not have any I2C hardware. A simulation using two GPIO lines is used, which is illustrated in Figure 28. Compared to Figure 3, the hardware connection of the I2C peripheral on the BSN node does not use peripheral memory. The condition of the peripheral module status does not influence the latency of the I2C sending function.

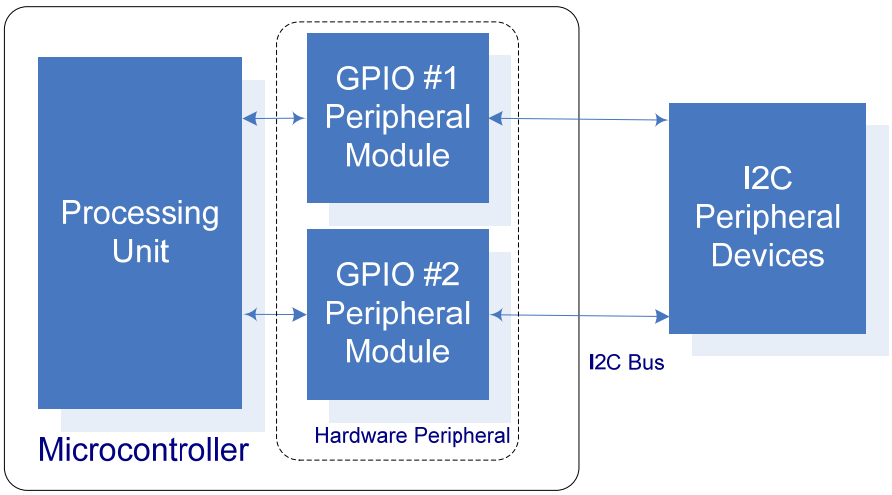


Figure 28 The hardware connection of the I2C peripheral on the BSN node

Another two conditions were concerned. One is the length of the sending queue. The other one is the baud rate of peripheral hardware. To investigate the relationship between the latencies and the two conditions, benchmarking exercises were carried

out where one of conditions varies, while at the same time, the other one was kept constant.

From Figure 26, it is observed that neither the length of the sending queue nor the baud rate influences the latency of peripheral function. The reason of this phenomenon on the CoolFlux development board can be referred in the UART section, as its I2C peripheral shares the same structure of hardware connection of peripherals with the UART peripheral. For the BSN node, the processing unit executes an I2C sending function, including an execution of GPIO sending functions eight times, to generate one I2C data byte. The range of the baud rate is limited, between 0 bps to 10 bps. The testing baud rate in the measurement, from 10 bps to 50 bps, does not influence the latency of the I2C sending function.

Latency of context switch

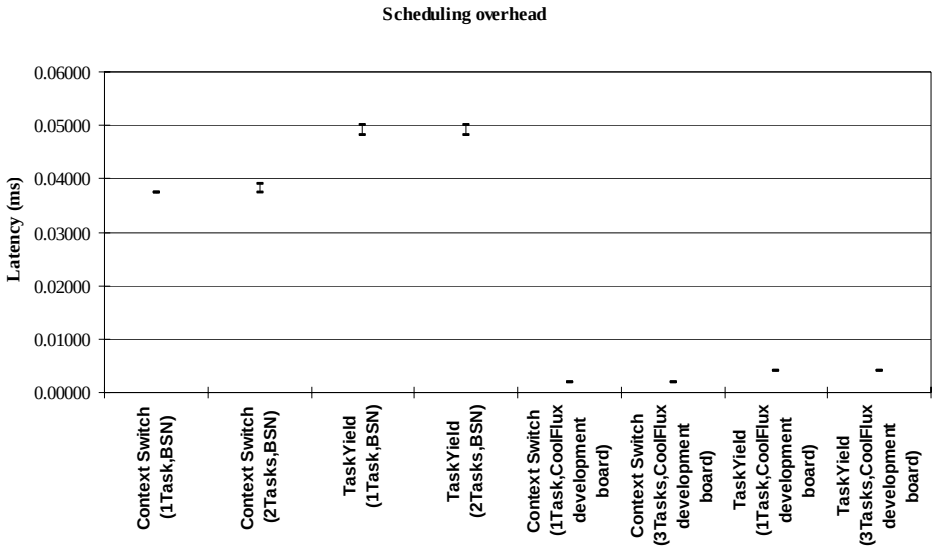


Figure 29 Latencies of context switch

The latencies of context switch are benchmarked by measuring *vTaskIncrementTick*, *vTaskSwitchContext* and *taskYIELD* functions. *vTaskIncrementTick* and *vTaskSwitchContext* functions are the main components of a context switch. *taskYIELD* is a macro for forcing a context switch. Measurements of

context switch were carried out when the scheduler generates a tick and schedules back to the same task. This is implemented by measuring those three context switch functions in the benchmarking task which has the highest priority. The multiple-executable measurement is used in context switch functions. From Figure 29, it is observed that the context switch time is constant if no other tasks are preempted.

6.3.3 Throughput

For applications of the benchmarking scope in section 1.4, communication among nodes and the communication between the node and its external devices are main tasks. The throughput of peripheral is one of the main metrics. A number of conditions for throughput were examined.

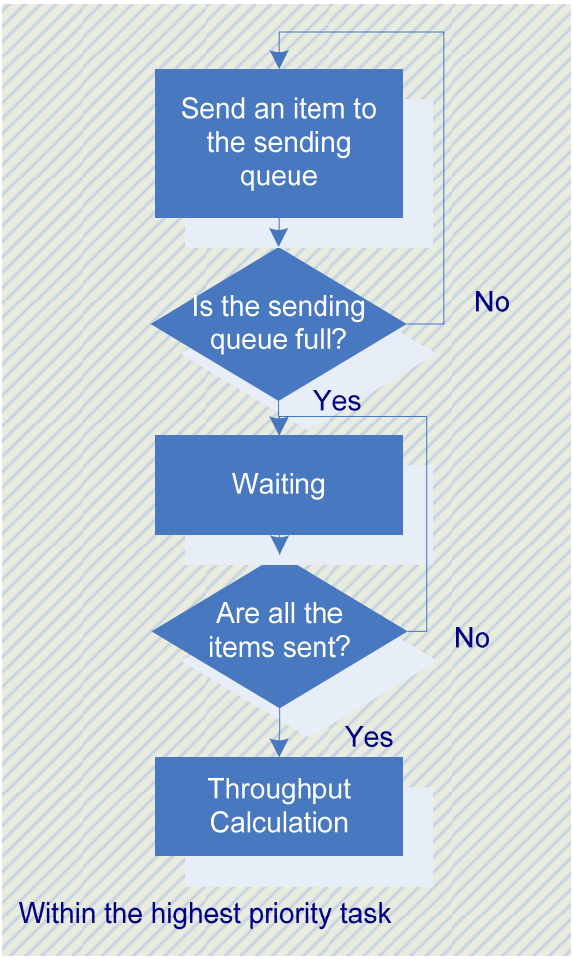


Figure 30 The process of throughput measurement

Peripherals with sending queues of specific length are created. The benchmarking task sends the same number of items to the sending queue as the length of the sending queue and waits until all the items are sent. The whole process is carried out when the sending task is given the highest priority. The duration time of the process of sending divided by the number of items sent is the throughput. The process can be illustrated in Figure 30.

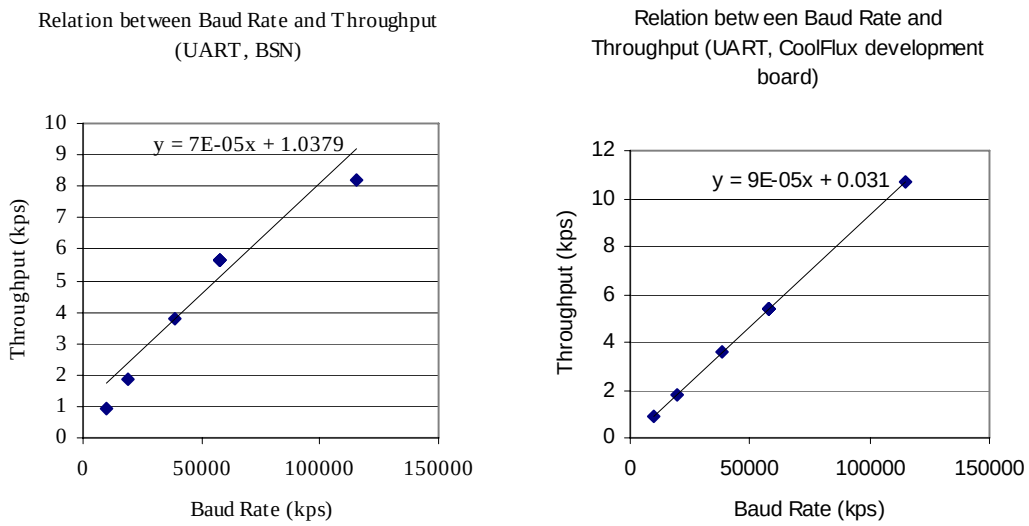


Figure 31 Throughput of UART sending

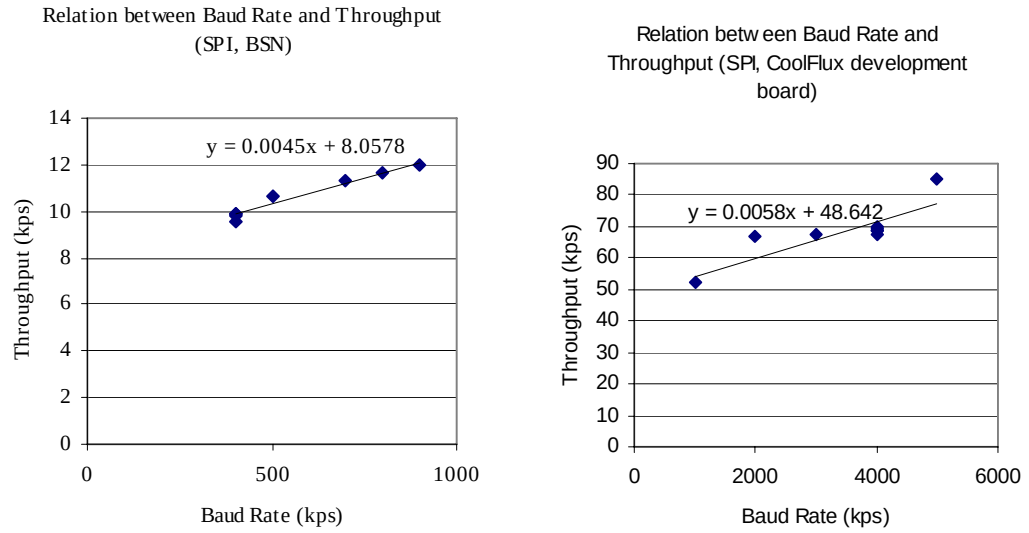


Figure 32 Throughput of SPI sending

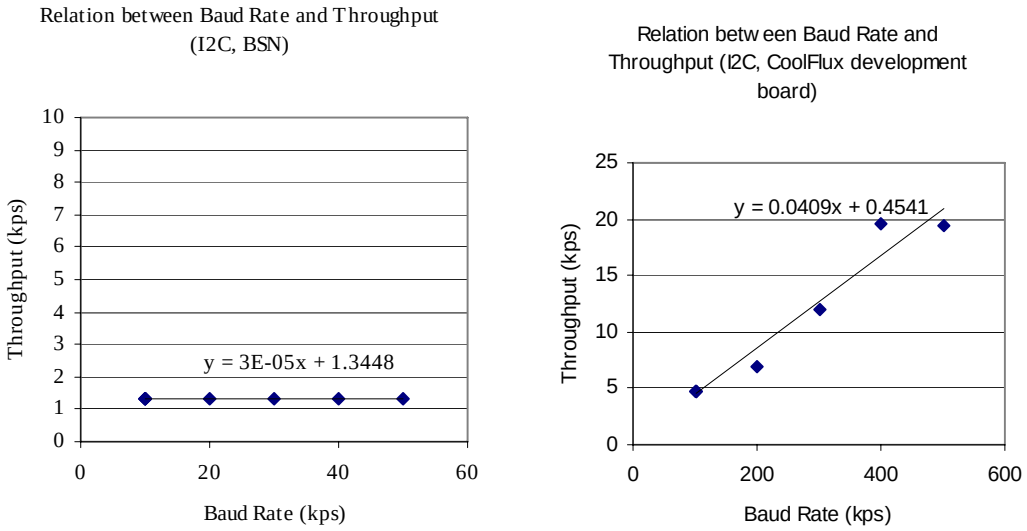


Figure 33 Throughput of I2C sending

As shown in Figure 31, Figure 32 and Figure 33, the throughput is dependent on the baud rate for a given peripheral on a given hardware platform. The throughput increases when the baud rate increases. The relationship between them is linear except the I2C peripheral on the BSN board. The linear relationship is expected, as the baud rate increases, the time for the hardware module sending one item decreases. In the Figure 32, the deviation of throughput to the linear function is large. This can be explained as the error of integer calculation. The values of peripheral registers are calculated by integers. As the baud rate increases, the values for baud rate registers decrease. When the baud rate reaches a certain value, the error of integer calculation is not negligible.

In Figure 33, there is an exception of linear relationship in the I2C module on the BSN board. This can be explained by follows. As discussed in section 1.1, the I2C module on the BSN board is implemented by using GPIO lines on the board to simulate the I2C protocol. As shown in Figure 24, the latencies of GPIO functions are not negligible. The latencies of GPIO functions dominate the throughput of the I2C peripheral of the BSN board.

6.3.4 Memory usage

A number of functions request heap spaces. Conditions and calculation equations are listed in this section. As discussed in section 4.4.2, measurement of memory usage is implemented by marking locations of heap space. The amount of memory usage is indicated by the difference of the starting heap mark and the ending heap mark of the heap.

Task creation

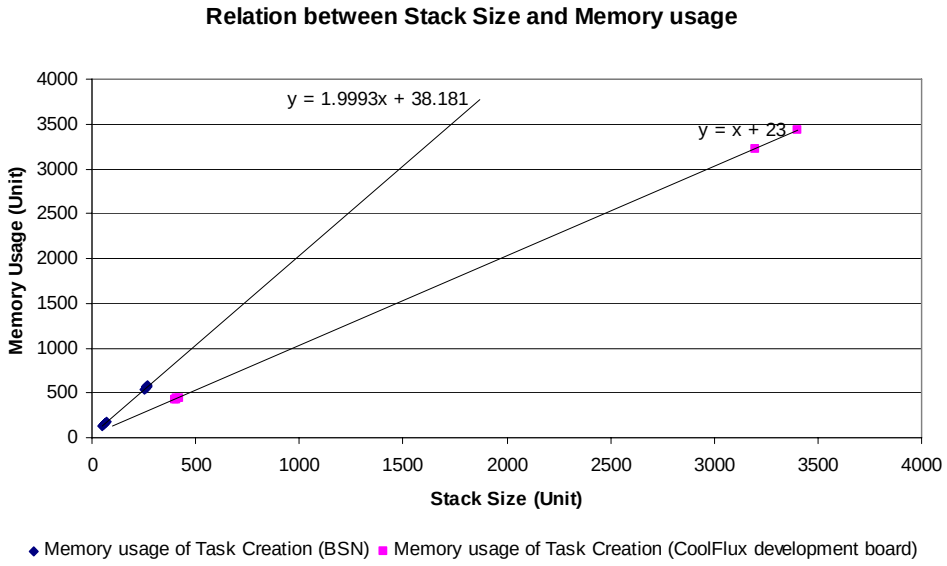


Figure 34 Relationship between stack size and memory usage

Task creation function requests a space for task handler and a space for the task content. The memory usage of the task handler is constant for a given processor. The memory usage of the task content is dependent on the stack size. The relationship between the stack size and memory usage is expected to be linear. In the benchmark stage, benchmark exercises were carried 13 times on the BSN node and 6 times on the CoolFlux development board. Figure 34 shows the linear relationship between stack size and memory usage for the BSN board and the CoolFlux development board.

It is also observed in Figure 34 that the amount of memory usage of the task creation function in the CoolFlux development board is a factor of two than in the BSN node. The BSN node and the CoolFlux development board have different memory organizations. The stack type is implemented as the integer type in both the BSN node and the CoolFlux development board. An integer datum occupies two memory bytes in the BSN node while in the CoolFlux development board it occupies one memory byte. For a given stack item, the BSN node allocates twice the number of memory bytes as the in the CoolFlux development board.

Semaphore and Queue

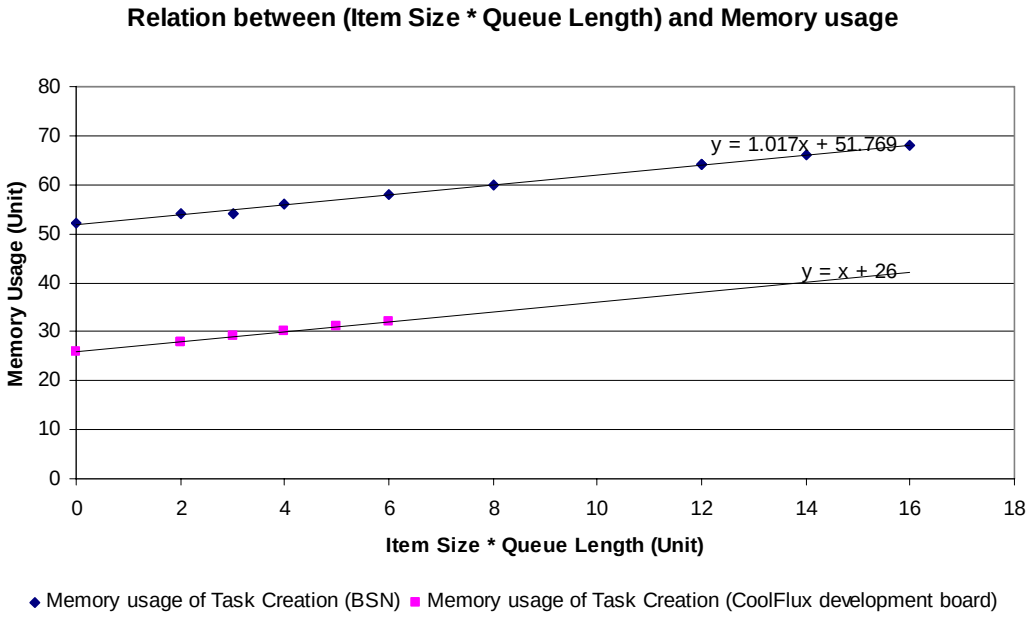


Figure 35 Relation between Item* Queue Length and memory usage

A semaphore is implemented by a queue structure. Both the semaphore and queue creation functions request space for semaphore/queue handlers and space for the semaphore/queue content. The memory usage of the semaphore/queue handler is constant. The memory usage of the semaphore/queue content is dependent on the length of the queue and the size of items. The relationship between the stack size and memory usage is expected to be linear. In the benchmark stage, benchmark exercises were carried 10 times on the BSN node and 7 times on the CoolFlux development

board. Memory usage of semaphores and queues is illustrated in Figure 35. Figure 35 shows the linear relationship between Item × Queue Length and memory usage.

Memory usage for UART, SPI and I2C creation

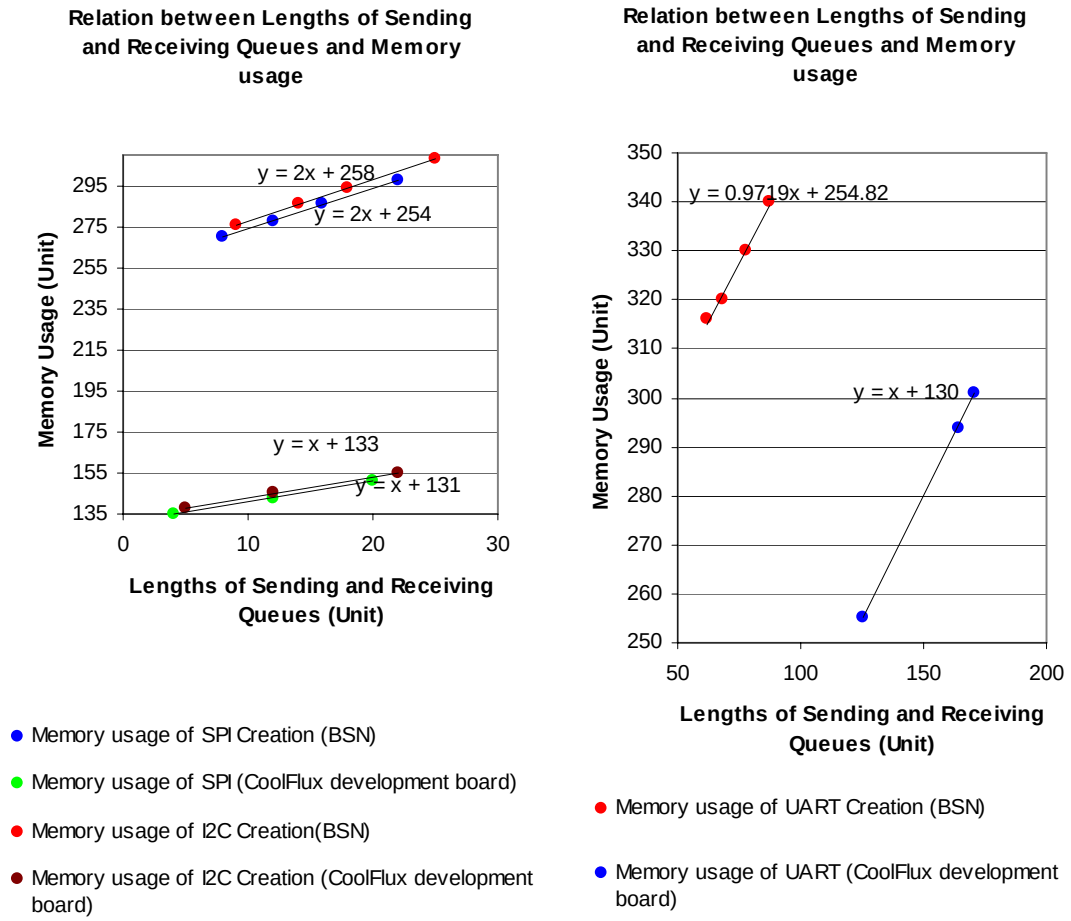


Figure 36 Relationship between length of sending and receiving queues and memory usage

Peripheral creation function requests a space for Peripheral handler and a space for the peripheral content. The memory usage of the Peripheral handler is constant. The memory usage of the Peripheral content is dependent on the stack size. The relationship between the stack size and memory usage is expected to be linear. In the benchmark stage, benchmark exercises were carried 4 times on the BSN node and 3 times on the CoolFlux development board for individual hardware module. Figure 36

shows the linear relationship between the length of the sending queue and memory usage for BSN board and CoolFlux development board.

It is also observed in Figure 36 that the amount of memory usage of the peripheral creation function in the CoolFlux development board is a factor of two larger than in the BSN node. The BSN node and the CoolFlux development board have different memory organizations. The stack type is implemented as the integer type in both the BSN node and the CoolFlux development board. An integer datum occupies two memory bytes in the BSN node while in the CoolFlux development board it occupies one memory byte. For a given item of peripheral content, the BSN node allocates twice the number of memory bytes as in the CoolFlux development board.

6.3.5 Code size

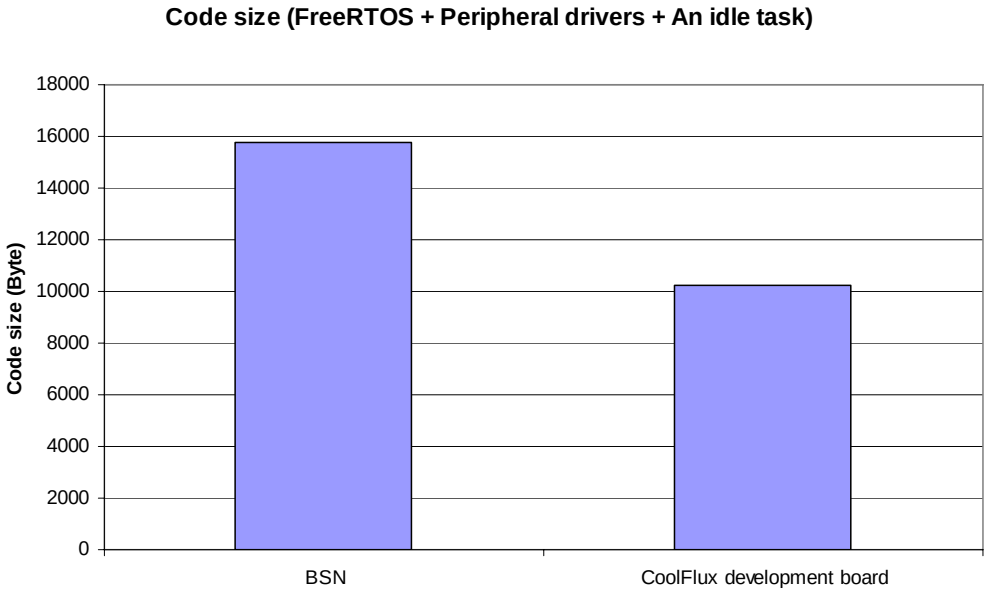


Figure 37 Code size (FreeRTOS + Peripheral drivers + An idle task)

Beside the RAM usage of the FreeRTOS and its hardware abstraction, their code size takes space. Figure 37 shows the code size of the FreeRTOS kernel, peripheral

drivers and an idle task. The amount of code size of the CoolFlux development board is less than the BSN node, which can be explained by the compiling technology of the CHESS compiler.

Chapter 7

Conclusion

The FreeRTOS kernel is a light kernel supporting a number of commonly used system functions. Its hardware abstraction contains popular peripheral drivers in the benchmarking scope (Section 1.4). As mentioned in section 1.2, this thesis consists of three objectives. As the first objective, metrics for benchmark are examined and identified in Chapter 3 and the measurement results are listed in Appendix C.

As the second objective of the thesis, a benchmark tool for benchmark was developed. The design issues, structure and the use of the tool are discussed in Chapter 5. As evidences of the configurability, the benchmark tool was configured individually to the BSN node and the CoolFlux development board as case studies. Hardware specific configuration and benchmark configuration were carried out. The configuration process is illustrated in 5.3.4. Take a look at the source code of the benchmark tool, the hardware specific configuration has 9 lines in the *measureTool.c* file for the BSN node while 21 lines for the CoolFlux development board, and 6 lines in the *benchmarkMain.h* file for both the BSN node and CoolFlux development board. The *benchmarkMain.c* file does not contain any configuration code. To sum up, the hardware specific configuration part accounts for 1.11% in the source code of the benchmark tool for the BSN node and 2.00% for the CoolFlux development board. The benchmark configuration only exists in the *benchmarkMain.h* file. It has 39 lines for

both the BSN node and the CoolFlux development board, which accounts for 2.89% in the source code of the benchmark tool.

As the third objective of the thesis, conditions influencing the performance of the FreeRTOS kernel and its hardware abstraction are investigated. The number of tasks does not influence the latency measurements in the benchmark tool. As illustrated in section 5.3.3, the metrics are benchmarked in the benchmark task with the highest priority. The benchmark task is not preempted by other tasks. The execution of functions in the benchmark tool is not influenced. The duration time of executing these functions is the same as if only one task is running in the kernel.

The latencies of a subset of functions are small, such as the Enter/Exit critical functions pair and peripheral sending functions with the condition of the sending queue is empty. These functions do not operate complex data structure and are executed sequentially which makes them easy to analyze from a static point of view. The execution of other functions takes relatively longer time, such as queues related functions, task handling functions and peripheral sending functions with the condition of the sending queue is not empty. These functions communicate with the task structure and queue structure, as discussed in 2.1.4, which are made suitable for event handling.

As discussed in section 6.1.1, the BSN node uses two GPIO modules to simulate an I2C module while the CoolFlux development board has dedicated I2C hardware. From Figure 33, the advantages of dedicated peripheral hardware are shown. The throughput of the I2C module of the BSN node is dominated by the latencies of the GPIO functions and is slower compared to the UART and SPI modules. During the whole course of I2C communication, the processor on the BSN node is occupied by the I2C simulation task, while for the CoolFlux development board, the processor can concentrate on another task when the I2C module is generating signals.

In the benchmarking exercises, the compiling technologies and computer architectures influence the performance. Both the BSN node and the CoolFlux development board share the similar source code for the system functions. From Figure 22 and Figure 23, it is shown that the scales among system calls latencies for the BSN node is different from the ones for the CoolFlux development board. E.g. the latency of queue sending function is 50.3 times larger than the latency of critical function pair in the BSN node, while in the CoolFlux development board, the scale is 23.3. This is because different compiling technologies compile the same source code into different machine code and processors, with different computer architectures, execute the same machine code with different duration.

Bibliography

1. (Aug 28, 2008). "GNU gprof." from http://sourceware.org/binutils/docs/gprof/index.html.
2. . "Linux Performance Analysis Tool: Oprofile." from http://oprofile.sourceforge.net/about/.
3. . "Linux Trace Toolkit ", from http://www.opersys.com/LTT/index.html.
4. Antoine Colin, I. P. (2001). "Worst-case Execution Time Analysis of the RTEMS Real-Time Operating System." IEEE Computer Society
5. Aoun, M. (2007). Distributed Task Synchronization inWireless Sensor Networks, RWTH Aachen. **Master**.
6. Barry, R. (2008, Jun 27, 2008). "Embedded System Performance Comparisons." from http://www.freertos.org/PC/index.html.
7. Barry, R. (2008). "FreeRTOS." from http://www.freertos.org.
8. Barry, R. (2008). "FreeRTOS FAQ " Retrieved Aug 11th, 2008, from http://www.freertos.org/FAQHelp.html.
9. Barry, R. (2008). "Legacy Trace Utility." Retrieved Jul 7, 2008, from http://www.freertos.org/a00086.html.
10. Barry, R. (2008). "Memory Management." from http://www.freertos.org/a00111.html.
11. Baynes, K., C. Collins, et al. (2001). The Performance and Energy Consumption of Embedded Real-Time Kernels. International Conference on Compilers, Architecture and Synthesis for Embedded Systems Atlanta, Georgia, USA, ACM.
12. Catalano, J. (2005). Implementation of sensor drivers for Small Autonomous Network Devices on the CoolFlux DSP. **Master:** 118.
13. Catalano, J. (2006). Operating system for CoolFlux DSP, Philips Research.
14. Catalano, J. (2008). FreeRTOS A Real Time Kernel for Wireless Sensor Network Platforms, Philips Research.
15. Cormac Duffy, U. R., John Herbert and Cormac J. Sreenan (2006). A Performance Analysis of MANTIS and TinyOS. Cork, University College Cork**:** 30.
16. Douglas C.Schmidt, M. D., Carlos O'Ryan (2002). Operating System Performance in Support of Real-time Middleware. Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems.
17. Eustace, A. and A. Srivastava (1995). ATOM: a flexible interface for building high performance program analysis tools. Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings New Orleans, Louisiana USENIX Association
18. Gopalakrishnan, R. (2005). Characterization of Linux on Philips Multimedia Platforms**:** 49.
19. IRISA. (2003, Dec 23, 2003). "Heptane Static WCET Analyzer." Retrieved Jun 28, 1008, 2008, from http://www.irisa.fr/aces/work/heptane-demo/heptane.html.

20. Larry McVoy, C. S. (1996). lmbench: Portable tools for performance analysis. Proceedings of the USENIX 1996 Annual Technical Conference.
21. Mueller, F. (2000). "Timing Analysis for Instruction Caches." Kluwer Academic Publishers **18**(2-3).
22. N.V., T. C. T. (2008). from http://www.retarget.com/.
23. Olusola, F. (2007). The evaluation of TinyOS with wireless sensor node kernels. Computer Systems Engineering, Halmstad University. **Master:** 79.
24. Park, S., J. W. Kim, et al. (2006). Embedded Sensor Networked Operating System. Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, IEEE Computer Society
25. Philips (2005). C Programmer's Manual**:** 64.
26. Philips (2006). CoolFlux DSP Demonstrator Chip Data Sheet.
27. Philips. (2008). "Ambient Intelligence - Changing lives for the better." from http://www.research.philips.com/technologies/syst_softw/ami/background.html.
28. Preusker, N. (2008). Viability of a real time operating system for the 8051 architecture, Technische Fachhochschule Berlin.
29. Puschner, P. and C. Koza (1989). "Calculating the Maximum Execution Time of Real-Time Programs." Kluwer Academic Publishers
30. Schoofs, A. (2006). 802.15.4 MAC for the SAND platform A new dedicated architecture for CoolFlux, Philips Research.
31. TechNet, M. (2005). "HyperTerminal overview." Retrieved Jul 2008, 2008, from http://technet2.microsoft.com/windowsserver/en/library/02c2459f-5b84-45fb-afab-610374d359941033.mspx?mfr=true.
32. TexasInstruments (2006). MSP430x1xx Family User's Guide.
33. Underwood, S. (2003). mspgcc A port of the GNU tools to the Texas Instruments MSP430 microcontrollers**:** 21.
34. Weiss, K., T. Steckstor, et al. (1999). Performance Analysis of a RTOS by Emulation of an Embedded System. Proceedings of the Tenth IEEE International Workshop on Rapid System Prototyping IEEE Computer Society
35. Westwood, P. (2006, Oct 15th, 2006). "Cold Cache ", from http://blog.ftwr.co.uk/wordpress/wp-cache-inspect/.
36. Wikipedia. (2008). "Benchmark (computing)." from **http://en.wikipedia.org/wiki/Benchmark_(computing)**.
37. Wikipedia. (2008). "Semaphore (programming)." from http://en.wikipedia.org/wiki/Semaphore_(programming).

Appendix A
FreeRTOS API

This description corresponds to the metrics of system functions in section 4.4, providing fundamental introduction of these functions. This section briefly introduces the FreeRTOS API. The interested reader can refer to FreeRTOS website (Barry 2008) and to the source code itself for more details. This section is intended as a reference for benchmarking system functions.

A.1 Task Creation

xTaskCreate()

This function creates a new task and adds it to the list of tasks that are ready to run.

Parameters:

pvTaskCode Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).

pcName A descriptive name for the task. This is mainly used to facilitate debugging.

usStackDepth The size of the task stack specified as the number of data of a given depth - not the number of bytes. For example, if the stack is 16 bits wide and *usStackDepth* is defined as 100, 200 bytes will be allocated for stack storage. The stack depth multiplied by the stack width must not exceed the maximum value that can be contained in a variable of type *size_t*.

pvParameters Pointer that will be used as the parameter for the task being created.

uxPriority The priority at which the task should run.

pvCreatedTask Used to pass back a handle by which the created task can be referenced.

A.2 Task Control

vTaskDelay()

This function delays a task for a given number of ticks. The actual time that the task remains blocked depends on the tick rate.
vTaskDelay() specifies a time at which the task wishes to unblock relative to the time at which *vTaskDelay*() is called. For example, specifying a block period of 100 ticks will cause the task to unblock 100 ticks after *vTaskDelay*() is called.
Parameters:
xTicksToDelay The amount of time, in tick periods, that the calling task should block.

A.3 Kernel Control

taskYIELD()
This is a macro for forcing a context switch.

taskENTER_CRITICAL()/taskEXIT_CRITICAL()

This is a macro to mark the start/end of a critical code region. Preemptive context switches cannot occur when in a critical region.

taskDISABLE_INTERRUPTS() / taskENABLE_INTERRUPTS()

This is a macro to disable/enable all maskable interrupts.

A.4 Queue Management

xQueueCreate()

This function creates a new queue instance. This allocates the storage required by the new queue and returns a handle for the queue.

Parameters:

uxQueueLength The maximum number of items that the queue can contain.

uxItemSize The number of bytes each item in the queue will require. Items are queued by copy, not by reference, so this is the number of bytes that will be copied for each posted item. Each item on the queue must be the same size.

xQueueSend()

This function posts an item on a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine.

Parameters:

xQueue The handle to the queue on which the item is to be posted.

pvItemToQueue A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so the corresponding number of bytes will be copied from *pvItemToQueue* into the queue storage area.

xTicksToWait The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to zero.

xQueueReceive()

This function receives an item from a queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

Parameters:

pxQueue The handle to the queue from which the item is to be received.

pvBuffer Pointer to the buffer into which the received item will be copied.

xTicksToWait The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call.

A.5 Semaphores

vSemaphoreCreateBinary()

This function is a macro that creates a semaphore by using the existing queue mechanism. The queue length is 1 as this is a binary semaphore. The data size is 0 as we don't want to actually store any data - we just want to know if the queue is empty or full.

Parameters:

xSemaphore Handle to the created semaphore.

xSemaphoreGive()

This function is a macro to release a semaphore. The semaphore must have previously been created with a call to *vSemaphoreCreateBinary*(), and obtained using *sSemaphoreTake*().

Parameters:

xSemaphore A handle to the semaphore being released. This is the handle returned when the semaphore was created.

xSemaphoreTake()

This function is a macro to obtain a semaphore. The semaphore must have previously been created with a call to vSemaphoreCreateBinary(),.

Parameters:

xSemaphore A handle to the semaphore being taken - obtained when the semaphore was created.

xBlockTime The time in ticks to wait for the semaphore to become available. A block time of zero can be used to poll the semaphore.

Appendix B

Hardware abstraction API

This description corresponds to the metrics of peripheral functions in section 4.5, providing the introduction of these functions.

B.1 GPIO (General Purpose Input Output)

xGpioPeripheralInit

This function reserves and configures the GPIO lines.

Parameters:

ePortMask A mask of the GPIO ports to include in this peripheral handle

pcPeripheralName The name of the GPIO peripheral

eDirectionMask The direction of the masked *eGPIOLines*

eInterruptEnableMask The mask for the GPIO ports to be initialized with interrupt enabled

ePolarityMask The mask for the settings of the interrupt polarity of the GPIO ports

vGpioPeripheralSend

This function sets the GPIO lines used in the *PeripheralHandle*.

Parameters:

pxPort The *PeripheralHandle* of the *GPIOLines* used in this handle

xData Every bit position represents an *eGPIOLine*, which will be set to the bit value if et to output and if included in the *PeripheralHandle*

xGpioPeripheralReceive

This function sets the GPIO lines used in the *PeripheralHandle*.

Parameters:

pxPort The *PeripheralHandle* of the *GPIOLines* used in this handle

B.2 UART (Universal Asynchronous Receiver-Transmitter)

xUartPeripheralInit

This function reserves and configures the UART interface.

Parameters:

pxPort The UART port

pcPeripheralName The name of the UART peripheral

eBaudrate The requested baud rate, selected in the *eUartBaudrate* enumeration

eParity The parity settings, selected in the *eUartParity* enumeration

eDirection The direction of the port, selected in the *eUartDirection* enumeration

uxTxQueueLength The *Tx* queue length

uxRxQueueLength The *Rx* queue length

xUartPeripheralReceive

This function receives a byte through the UART interface.

Parameters:

pxPort The UART port, initialized by *xUartPeripheralInit*()

pcRxChar Points to the memory space where the received character will be written

xBlockTime The time in ticks to wait if the queue is empty.

xUartPeripheralSend

This function sends a byte through the UART interface.

Parameters:

pxPort The UART port, initialized by *xUartPeripheralInit*()

cTxChar The character to transmit

xBlockTime The time in ticks to wait if the queue is full.

vUARTTXISR

This ISR Handles the UART *Tx* interrupt.

Parameters:

ePort The peripheral port initialized by *xUartPeripheralInit*()

vUARTRXISR

This ISR Handles the UART Rx interrupt

Parameters:

ePort The peripheral port initialized by *xUartPeripheralInit*()

B.3 SPI (Serial Peripheral Interface)

xSpiPeripheralInit

This function configures the SPI interface and creates driver structure.

Parameters:

ePort The SPI port

pcPeripheralName The name of the SPI peripheral

xKiloBitRate The bit rate, in kilo bits /sec.

xSpiMode The SPI mode

uxTxQueueLength The *Tx* queue length

uxRxQueueLength The *Rx* queue length

xSpiPeripheralReceive

This function receives one byte of data from the Rx queue.

Parameters:

pxPort The SPI port, initialized by *xSpiPeripheralInit*()

pxRxWord The data send from the Rx queue.

pxContinue The status of the continue bit during the receive.

xBlockTime The time in ticks to wait if the queue is empty / Rx queue contains data.

xSpiPeripheralTransfer

This function makes a transfer through the SPI interface (The Rx data will be written directly in the *RxQueue* by the SPI ISR if *xReceive* is set *pdTRUE*).

Parameters:

pxPort The SPI port, initialized by *xSpiPeripheralInit*()

xTxWord The word to send over the SPI, to request data from a slave.

xContinue Set to one if more data has to be transferred in the same bunch. Otherwise, zero.

xReceive If set to '0' then received data will be ignored otherwise received data goes to rxQueue.

xBlockTime The time in ticks to wait if the queue is empty / Rx queue contains data.

vSPIISR

This ISR Handles the SPI interrupt

Parameters:

ePort The peripheral port initialized by *xSpiPeripheralInit* ()

B.4 I2C (Inter-Integrated Circuits)

xI2cInitQueueReceive

This function initiates receiving bytes from the I2C interface into the Rx queue.

Parameters:

pxPort The I2C port, initialized by xI2CPeripheralInit()

xSlaveAddress The I2C address of the sending slave

xBytesToReceive The number of bytes to receive into the *RxQueue*

xRestart if *pdTRUE*, a restart condition is sent on the bus before starting this receiving.

xStop if *pdTRUE*, a stop condition is sent on the bus after this receiving.

xBlockTime The time in ticks to wait if the queue is full.

xI2cPeripheralInit

This function reserves and configures the I2C interface.

Parameters:

ePort The I2C port

pcPeripheralName The name of the I2C peripheral

xKiloBitRate The bit rate, in kilo bits /sec.

xPeripheralEnable Status of the I2C bus (*pdTRUE* for enabled, *pdFALSE* for disabled).

uxTxQueueLength The Tx queue length

uxRxQueueLength The Rx queue length

xI2cPeripheralReceive

This function receives a byte from the I2C Rx queue.

Parameters:

pxPort The I2C port, initialized by xI2CPeripheralInit()

pxSlaveAddress Points to the memory space where the I2C address of the sending slave will be written

pxRxByte Points to the memory space where the received word will be written

xBlockTime The time in ticks to wait if the queue is empty.

xI2cPeripheralSend

This function sends a byte through the I2C interface.

Parameters:

pxPort The I2C port, initialized by xI2CPeripheralInit()

xSlaveAddress The I2C address of the receiving slave

xTxByte The byte to transmit

xRestart If *pdTRUE*, a (re)-start condition will be issued on the bus before sending

xSlaveAddress (again)

xStop If *pdTRUE*, a stop condition will be issued on the bus after sending *xTxByte*

xBlockTime The time in ticks to wait if the queue is full.

vI2CISR

This ISR Handles the I2C interrupt.

Parameters:

ePort The peripheral port initialized by xI2CPeripheralInit ()

Appendix C

Tables of Results

Table 3 Latency of Task Creation (1st call, BSN node)

1stTaskCreation	Depth	Execution Time (ms)	Expected Time (ms)	Deviation (%)
Sample1	250	0.77368	0.77369	0.00
Sample2	257	0.78640	0.78635	0.01
Sample3	50	0.41368	0.41189	0.43
Sample4	55	0.42089	0.42094	0.01
Sample5	60	0.42998	0.42998	0
Sample6	66	0.44089	0.44083	0.01

Table 4 Latency of Task Creation (1st call, CoolFlux development board)

1stTaskCreation	Depth	Execution Time (ms)	Expected Time (ms)	Deviation (%)
Sample1	500	0.02608	0.02610	0.08
Sample2	520	0.02660	0.02650	0.37
Sample3	580	0.02785	0.02770	0.54
Sample4	590	0.02790	0.02790	0
Sample5	3950	0.09782	0.09510	2.78

Table 5 Latency of Task Creation (2nd call, BSN node)

2ndTaskCreation	Depth	Execution Time (ms)	Expected Time (ms)	Deviation (%)
Sample1	260	0.68640	0.68824	0.27
Sample2	57	0.32089	0.32101	0.04
Sample3	64	0.33362	0.33368	0.02
Sample4	70	0.34453	0.34453	0

Table 6 Latency of Task Creation (2nd call, CoolFlux development board)

2ndTaskCreation	Depth	Execution Time (ms)	Expected Time (ms)	Deviation (%)
Sample1	540	0.02191	0.02190	0.04
Sample2	560	0.02235	0.02230	0.22
Sample3	600	0.02318	0.02310	0.35
Sample4	620	0.02349	0.02350	0.04

Table 7 Latency of Task Creation (3rd call, BSN node)

3rdTaskCreation	Depth	Execution Time (ms)	Expected Time (ms)	Deviation (%)
Sample1	263	0.69362	0.69367	0.01
Sample2	268	0.70271	0.70271	0.00
Sample3	270	0.70634	0.70633	0.00

Table 8 Latency of Task Creation (3rd call, CoolFlux development board)

3rdTaskCreation	Depth	Execution Time (ms)	Expected Time (ms)	Deviation (%)
Sample1	3600	0.08541	0.08560	0.22
Sample2	3650	0.08652	0.08660	0.09
Sample3	3800	0.08968	0.08960	0.08
Sample4	3850	0.09065	0.09060	0.06

Table 9 System call overhead (BSN node)

	Average (ms)	Max (ms)	Min (ms)	Deviation (%)
Critical	0.00289	0.00459	0.00277	58.8
Semaphore Create	0.21488	0.21549	0.21368	0.56
Queue Create	0.13186	0.13186	0.13186	0
Semaphore Give	0.08656	0.08822	0.08640	1.92
Semaphore Take	0.08480	0.08640	0.08459	1.89
Queue Send	0.14538	0.14822	0.14277	1.95

Queue Receive	0.14256	0.14640	0.13913	2.69

Table 10 System call overhead (CoolFlux development board)

	Average (ms)	Max (ms)	Min (ms)	Deviation (%)
Critical	0.00025	0.00033	0.00025	32.0
Semaphore Create	0.01154	0.01158	0.01149	0.78
Queue Create	0.00627	0.00633	0.00625	0.96
Semaphore Give	0.00539	0.00541	0.00533	1.11
Semaphore Take	0.00525	0.00533	0.00525	1.52
Queue Send	0.00582	0.00616	0.00575	5.84
Queue Receive	0.00574	0.00608	0.00566	5.92

Table 11 Latency of Gpio peripheral functions (BSN node)

	Average (ms)	Max (ms)	Min (ms)	Deviation (%)
Gpio Create	32.44640	32.44640	32.44640	0
Gpio Send	0.01612	0.01731	0.01549	7.38
Gpio Receive	0.00580	0.00640	0.00459	20.9

Table 12 Latency of Gpio peripheral functions (CoolFlux development board)

	Average (ms)	Max (ms)	Min (ms)	Deviation (%)
Gpio Create	0.03033	0.03035	0.03032	0.10
Gpio Send	0.00041	0.00050	0.00041	21.9
Gpio Receive	0.00039	0.00050	0.00033	28.2

Table 13 Latency of I2c peripheral functions (BSN node)

#Tasks	Send Queue Length	Baud Rate	Send (Send Queue Empty)	Send (Send Queue Not Empty	Throughput Latency

1	7	10	0.73637	0.73825	5.21186
1	11	10	0.73614	0.73819	8.17913
1	13	10	0.73590	0.73818	9.66095
1	16	10	0.73598	0.73821	11.88640
1	16	20	0.73605	0.73816	11.88458
1	16	30	0.73609	0.73818	11.88459
1	16	50	0.73612	0.73816	11.88640
1	16	40	0.73616	0.73809	11.88641

Table 14 Latency of I2c peripheral functions (CoolFlux development board)

I2C	Send Queue Length	Baud Rate	Send (Send Queue Empty)	Send (Send Queue Not Empty	Throughput Latency
3Tasks	100	9	0.00750	0.00658	1.90041
3Tasks	100	11	0.00752	0.00657	2.31952
3Tasks	100	7	0.00752	0.00659	1.47943
3Tasks	100	3	0.00749	0.00657	0.64015
1Task	100	17	0.00749	0.00656	3.58357
1Task	200	40	0.00749	0.00657	5.83474
1Task	300	30	0.00752	0.00657	2.48585
1Task	400	20	0.0075	0.00657	1.01741
1Task	500	10	0.0075	0.00657	0.51258

Table 15 Latency of Spi peripheral functions (BSN node)

#Tasks	Send Queue Length	Baud Rate	Send (Send Queue Empty)	Send (Send Queue Not Empty	Throughput Latency
1	5	400	0.06221	0.11407	0.52095
1	8	400	0.06210	0.11399	0.81731
1	11	400	0.06221	0.11410	1.11549
1	15	400	0.06224	0.11378	1.51368
1	15	500	0.05548	0.11381	1.41185
1	15	700	0.05004	0.11385	1.33004
1	15	900	0.04459	0.11408	1.24822

1	15	800	0.04731	0.11398	1.28823

Table 16 Latency of Spi peripheral functions (CoolFlux development board)

SPI	Send Queue Length	Baud Rate	Send (Send Queue Empty)	Send (Send Queue Not Empty	Throughput Latency
3Tasks	4000	9	0.00200	0.00639	0.12916
3Tasks	4000	10	0.00202	0.00643	0.14385
3Tasks	4000	12	0.00202	0.00643	0.17393
3Tasks	4000	14	0.00199	0.00640	0.20365
1Task	4000	45	0.00199	0.00640	0.66682
1Task	1000	60	0.00199	0.00637	1.1539
1Task	2000	50	0.00202	0.00643	0.7486
1Task	3000	40	0.002	0.00641	0.59083
1Task	5000	30	0.002	0.00641	0.35175

Table 17 Latency of Uart peripheral functions (BSN node)

#Tasks	Send Queue Length	Baud Rate	Send (Send Queue Empty)	Send (Send Queue Not Empty	Throughput Latency
1	60	57600	0.01933	0.10727	10.59186
1	65	57600	0.01934	0.10729	11.47913
1	73	57600	0.01939	0.10728	12.89913
1	80	57600	0.01937	0.10732	14.13731
1	80	9600	0.01937	0.10727	84.50276
1	80	19200	0.01922	0.10730	42.32095
1	80	38400	0.01932	0.10732	21.21186
1	80	115200	0.01952	0.10739	9.76096

Table 18 Latency of Uart peripheral functions (CoolFlux development board)

UART	Send Queue Length	Baud Rate	Send (Send Queue Empty)	Send (Send Queue Not Empty	Throughput Latency

3Tasks	57600	60	0.00125	0.00608	11.21375
3Tasks	57600	80	0.00127	0.00610	14.95302
3Tasks	57600	100	0.00127	0.00610	18.68635
3Tasks	57600	120	0.00124	0.00607	22.42532
1Task	57600	89	0.00124	0.00607	16.63282
1Task	9600	90	0.00124	0.00607	99.83632
1Task	19200	95	0.00127	0.00610	52.6896
1Task	38400	110	0.00125	0.00608	30.51150
1Task	115200	120	0.00125	0.00608	11.25016

Table 19 Context switch time (BSN node)

	Average (ms)	Max (ms)	Min (ms)	Deviation (%)
Context Switch	0.03753	0.03913	0.03731	4.26
Task Yield	0.04940	0.05004	0.04822	2.39

Table 20 Context switch time (CoolFlux development board)

	#Tasks	Average (ms)	Max (ms)	Min (ms)	Deviation (%)
Context Switch	1	0.00182	0.00190	0.00182	4.39
Task Yield	1	0.00407	0.00407	0.00407	0
Context Switch	3	0.00183	0.00191	0.00183	4.37
Task Yield	3	0.00408	0.00408	0.00408	0

Table 21 Task Delay duration time

	Average (ms)	Max (ms)	Min (ms)	Deviation (%)
Task Delay(BSN)	9.77590	9.92277	9.45368	3.29
Task Delay(COOLFLUX DEVELOPEMENT BOARD)	9.98583	9.99416	9.53075	4.56

Table 22 Memory usage of Task creation (BSN board)

Stack Size	Memory
50	138
55	148
57	152
60	158
64	167
66	170
70	178
250	538
257	552
260	558
263	564
268	574
270	578

Table 23 Memory usage of Task creation (CoolFlux development board)

Depth	Memory Usage
400	423
405	428
410	433
420	443
3200	3223
3400	3423

Table 24 Memory usage of Semaphore/Queue creation (BSN board)

Item Type	Item Size	Length	Heap Usage
NULL	0	1	52
char	1	2	54
char	1	4	56
char	1	3	54
short	2	3	58

short	2	6	64
short	2	7	66
long	4	4	68
long	4	2	60
long	4	3	64

BSN The item sizes of data types can be found in the C language manual of BSN(Underwood 2003)

Table 25 Memory usage of Semaphore/Queue creation (CoolFlux development board)

Item Type	Item Size	Queue Length	Memory Usage
NULL	0	1	26
Char	1	3	29
Char	1	4	30
Short	1	2	28
Short	1	5	31
Long	2	1	28
Long	2	3	32

The item sizes of data types can be found in the C language manual of CoolFlux (Philips 2005).

Table 26 Memory usage of UART creation (BSN node)

Send Queue Length	Receive Queue Length	Heap Usage
60	2	316
65	3	320
73	5	330
80	7	340

Table 27 Memory usage of UART creation (CoolFlux development board)

Send	Receive	Memory Usage

Queue Length	Queue Length	
120	5	255
155	9	294
160	11	301

Table 28 Memory usage of SPI creation (BSN node)

Send Queue Length	Receive Queue Length	Heap Usage
5	3	270
8	4	278
11	5	286
15	7	298

Table 29 Memory usage of SPI creation (CoolFlux development board)

Send Queue Length	Receive Queue Length	Memory Usage
2	2	135
5	7	143
9	11	151

Table 30 Memory usage of I2C creation (BSN node)

Send Queue Length	Receive Queue Length	Heap Usage
7	2	276
11	3	286
13	5	294
16	9	308

Table 31 Memory usage of I2C creation (CoolFlux development board)

Send Queue Length	Receive Queue Length	Memory Usage
2	3	138
5	7	145
9	13	155

Table 32 Memory usage of GPIO creation (BSN node and CoolFlux development board)

Hardware platform	Memory usage
BSN node	88
CoolFlux development board	48

Appendix D

Configurable file of the benchmark tool (benchmarkMain.h)

```
#include "FreeRTOS.h"
#include "peripheral.h"
#include "task.h"
#include "drv_i2c.h"
#include "drv_uart.h"
#include "drv_gpio.h"
#include "drv_spi.h"
#include "drv_timer.h"
#include "clockshop.h"
#include <stdlib.h>
#include <string.h>
#include "list.h"
#include "queue.h"

#define STARTDEBUG                              0

/* Benchmark Module Selection */
#define BENCHMARK_BSN
                        1
#define BENCHMARK_COOLFLUX
                        0

#define BENCHMARK_GPIO_MODULE_SWITCH    0
#define BENCHMARK_SYS_MODULE_SWITCH
            0
#define BENCHMARK_BEH_MODULE_SWITCH
            0
#define BENCHMARK_ISR_MODULE_SWITCH
            1
#define BENCHMARK_THR_MODULE_SWITCH
            1

#define BENCHMARK_SPI_MODULE_SWITCH     1
#define BENCHMARK_I2C_MODULE_SWITCH     0

#define LENGTH_UART_TXBUF                       80
#define LENGTH_SPI_TXBUF                        15
#define LENGTH_I2C_TXBUF                        16

#define LENGTH_UART_RXBUF                       7
#define LENGTH_SPI_RXBUF                        7
#define LENGTH_I2C_RXBUF                        9

#define TEST_BAUD_RATE      (       eUartBaudrate       )
( ( unsigned portLONG ) 57600 )
#define SPI_BAUD_RATE       800
#define I2C_BAUD_RATE       40

/* BSN configuration */
#if BENCHMARK_BSN == 1
#define BINARY_DIGIT_SIZEOF_portHEAP_UNIT
            1 /* sizeof (portHEAP_UNIT) = 0B00..001, note
shift operation @ CoolFlux is a signed shift */
#define UART_TEST
                                UART1
#define TIMERCOUNTUP
                                1 /* The timer of MSP430
is in count up mode */
/* Hardware Configuration for Measurement */
#define SPI_TEST            SPI0
#define I2C_TEST            I2C0

#define TIMERPERIOD                     0xFFFF
#define TIMERDIVIDER                    8


#define                 portMEASURE_TIMER_CLOCK_HZ
        portCPU_CLOCK_HZ
#define portMEASURE_RESULT
            unsigned portLONG
```

```
#define portMEASURE_TIMER
        unsigned portSHORT
#define portHEAP_UNIT                   unsigned
portCHAR  /* Defined by FreeRTOS in Heap_1.c */
#define portPOINTER
        unsigned portBASE_TYPE

#define LENGTH_PRINT_BUF                        50

/* Debug options */
#define SHOW_EACH_DIGIT     0

/* Measurement Factors */
#define                 configBenchmark_STACK_SIZE
configMINIMAL_STACK_SIZE * 5
#define MAXMEASUREVALUE
        ( portMEASURE_RESULT )( 0x7FFFFFFF )
#define MINMEASUREVALUE
        ( portMEASURE_RESULT ) 0

#endif
/* COOLFLUX configuration */
#if  BENCHMARK_COOLFLUX == 1
#define BINARY_DIGIT_SIZEOF_portHEAP_UNIT
        1 /* sizeof (portHEAP_UNIT) = 0B00..001, note
shift operation @ CoolFlux is a signed shift */
#define UART_TEST
                        UART0
#define TIMERCOUNTUP
                        0 /* The timer of Coolflux
is in count down mode */
/* Hardware Configuration for Measurement */
#define SPI_TEST        SPI1
#define I2C_TEST        I2C0

#define TIMERPERIOD             0x7FFFFF
#define TIMERDIVIDER            1
#define LED_RED                 0   /*   Red
LED, GPIO 0 */
#define LED_GREEN               5 /* GreenLED,
GPIO 5 */
#define TEST_GPIO               9/*  Used  for
testing*/
#define PWR_GPIO14              14
#define PWR_GPIO15              15

#ifndef     portCPU_CLOCK_HZ /* Clock frequency of the
processor */
#define                         portCPU_CLOCK_HZ
        ( DeviceMainClock_kHz[Oscillator0] * 1000 )
#endif
#define             portMEASURE_TIMER_CLOCK_HZ
        portCPU_CLOCK_HZ
```

```
#define portMEASURE_RESULT
        unsigned portLONG
#define portMEASURE_TIMER
        unsigned portSHORT
#define portHEAP_UNIT                   unsigned
portCHAR  /* Defined by FreeRTOS in Heap_1.c */
#define portPOINTER
        unsigned portBASE_TYPE

/* Debug options */
#define SHOW_EACH_DIGIT     0
/* Measurement Factors */
#define configBenchmark_STACK_SIZE     3950
#define MAXMEASUREVALUE
        ( portMEASURE_RESULT )( 0x7FFFFF )
#define MINMEASUREVALUE
        ( portMEASURE_RESULT ) 0

#endif

/* Measurement Factors */
#define NUM_SAMP
        100 /* For a full test, it should be > 100*/

#define NUM_TEST_TASKS_STACKSIZEDEM
        0/*  Number  of  Sample  Tasks  with  various
Depth*/
#define SAMPLETASK_PRIORITY
                0
#define BENCHMARK_PRIORITY
                2
#define NUM_TESTITERATION_QUEUECREATE
        3 /* For a full test, the number should be 7 */
#define NUM_TEST_TASKS_COND

        NUM_TEST_TASKS_STACKSIZEDEM  +  1  /*
Several Sample Tasks + A measurement Task,  For a full
test, it should be 5*/
#define NUM_TEST_TASKYIELD_COND     5

#define NUM_TEST_QUEUE_LENGTH       5

#define NUM_TEST_QUEUE_TYPE                 3  /*
Type =  Char, Short, Long */


/* Measurement tools */
#define DEBUGFLAG 1
#define TESTCHAR   '$'
#define TESTSHORT  11
#define TESTLONG   22
#define TESTBASE   33

#define DIGIT_RESOLUTION_AF_SEC             8
```

```
#define DEC_RESOLUTION_AF_SEC
        100000000 /* The resolution by Second is 10^(-
DIGIT_RESOLUTION_AF_SEC)*/
#define DIGIT_RESOLUTION_AF_MS
        ( DIGIT_RESOLUTION_AF_SEC - 3 )
#define DEC_RESOLUTION_AF_MS
        100000 /* The resolution by Mini-Second is
10^(-DIGIT_RESOLUTION_AF_MS)*/
#define LENGTH_PRINTRESULTBUF                    11
/* The max length of results */
#define PrintMeasureResult_LARGEST_ONE  1000000000
/* 10^9, Max decimal denoted by 'unsigned long' with max
value of 2^32 = 4.3 * 10^9 */
#define PrintMeasureResult_SMALLEST_ONE  0

#define tick_BLOCK
        portMAX_DELAY
#define INDEXNULL
        ('#' - '0') /* '#' =  '0' -13 */
#define NUM_DIR
                2

/* Fixed data of Connected devices*/

#define WHOAMI
#define LIS3L_ADDR      0x3A

/* Define data structures */



 typedef enum
{
        ItemType_Ref = 0,
        ItemType_Tmp = 1,
} ItemRefTmp;


 typedef enum
 {
         DirSend = 0,
         DirRecv = 1,
         QueuInit = 2
 } DirComm;

 typedef enum
{
        IndexArr_HeapPoint_StartPoint  = 0,
        IndexArr_HeapPoint_LastSampleTask        =
NUM_TEST_TASKS_STACKSIZEDEM,
        IndexArr_HeapPoint_BenchmarkTask         =
NUM_TEST_TASKS_STACKSIZEDEM + 1,
        IndexArr_HeapPoint_Scheduler             =
NUM_TEST_TASKS_STACKSIZEDEM + 2,
        IndexArr_HeapPoint_Tmp
         = NUM_TEST_TASKS_STACKSIZEDEM + 3,
        IndexArr_HeapPoint_LastIndex  =
IndexArr_HeapPoint_Tmp
 }  IndexArr_HeapPoint_RecordArray;/*  Format  of
HeapRecordArray:                         [StartPoint,
1stSampleTask,...,LastSampleTask,BenchmarkTask,
Scheduler,Tmp]*/


  typedef enum
 {
        StartFlag = 0,
        StopFlag = 1
 } MeasureRecordPoints;

  typedef struct TypeMeasureValues
{
         portMEASURE_TIMER TestPoint[NUM_DIR];
         portMEASURE_RESULT  CurrentResult;
         portMEASURE_RESULT  MaxResult;
         portMEASURE_RESULT  MinResult;
         portMEASURE_RESULT  AvgResult;
         portMEASURE_RESULT  IterationCounter;
 } MeasureValues;



/* Define Macro and functions */
#define CURRENT_HEAP_POINT
          ( portHEAP_UNIT * ) pvPortMalloc(1)
#if BENCHMARK_COOLFLUX == 1
#define  WAIT_UART_SEND_DONE()  PortBusyFlag  =
(( xUARTConfig  *  ) xTerm->pxConfig)->pxPortBusy;
while(PortBusyFlag==pdTRUE)
#define  WAIT_SPI_SEND_DONE()  PortBusyFlag  =
((  xSPIConfig  *  )  xTestSpi->pxConfig)-
>pxPortBusy;while(PortBusyFlag==pdTRUE){  PortBusyFl
ag  =  ((  xSPIConfig  *  )  xTestSpi->pxConfig)-
>pxPortBusy;}//while ( (xTestSpi_Config->pxPortBusy) ==
pdTRUE )
#define  WAIT_I2C_SEND_DONE()  PortBusyFlag  =
((  xI2CConfig  *  )  xTestI2c->pxConfig)-
>pxPortBusy;while(PortBusyFlag==pdTRUE){  PortBusyFl
ag  =  ((  xI2CConfig  *  )  xTestI2c->pxConfig)-
>pxPortBusy;}//while ( (xTestI2c_Config->pxPortBusy) ==
pdTRUE )
#endif
#if BENCHMARK_BSN == 1
#define       WAIT_UART_SEND_DONE()          while
(  ((xUARTConfig  *)(xTerm->pxConfig))->pxPortBusy  ==
pdTRUE )
```

```
#define WAIT_SPI_SEND_DONE()  while ( ((xSPIConfig *)(xTestSpi->pxConfig))->pxPortBusy == pdTRUE )
#define WAIT_I2C_SEND_DONE()  while ( ((xI2CConfig *)(xTestI2c->pxConfig))->pxPortBusy == pdTRUE )
#endif
#if BENCHMARK_BSN == 1
#define DISABLE_CONTEXT_SWITCH(Flag) if(Flag == StartFlag) TACCTL0 &=~CCIE; else TACCTL0 |= CCIE;
#endif
#if BENCHMARK_COOLFLUX == 1
#define DISABLE_CONTEXT_SWITCH(Flag)
#endif
#define SHOW_HEAP_USAGE_BY_PERCENT( Point_Heap_Usage, CharString, FlagChar )
        {vPrintMeasureResult( ( 100 * ( portLONG ) Point_Heap_Usage * DEC_RESOLUTION_AF_MS )/ configTOTAL_HEAP_SIZE,(char *) CharString,FlagChar + '0');WAIT_UART_SEND_DONE();}
#define SHOW_HEAP_USAGE_BY_UNIT( val, CharString, FlagChar ) {vPrintMeasureResult( ( portLONG ) ( val - Overhead_HeapMeasureFun) * DEC_RESOLUTION_AF_MS ,(char *) CharString,FlagChar + '0');WAIT_UART_SEND_DONE();}
#define POINT_HEAP_xNextFreeByte
        (( size_t )( CURRENT_HEAP_POINT - ( portHEAP_UNIT * ) pvHeapStartPoint ) >> ( BINARY_DIGIT_SIZEOF_portHEAP_UNIT - 1 ))/* Divided by sizeof(portHEAP_UNIT) */
/* Declaim functions */

void vBenchmarkProject(unsigned portSHORT uxPriority);
void vStartBenchmark( void *pvParameters );
void vSampleTask( void *pvParameters );
void vBenchmark_SysFun();
void vBenchmark_Drivers();
void vBenchmark_SysBehavior();
void vBenchmark_OtherFeatures();
void vBenchmark_Done();
void vInitMeasureValuables();
void INIT_RESULT_0D(MeasureValues *pMetrics_sample);
void INIT_RESULT_1D(MeasureValues * pMetrics_sample, portSHORT num_para);
void INIT_RESULT_2D();
void vTestOverheadMesureFun();
void vProcToResultStrcture(MeasureValues *pMyMeasureValues);
portMEASURE_RESULT vCalcTimerCount2MS( portMEASURE_TIMER TimerCountStart,  portMEASURE_TIMER TimerCountEnd);
void vBenchmark_SysBehavior();
void vPrintFullResults (MeasureValues myMeasureValues, char * NameMetricsString, portCHAR indexNumber);
void vShowHeapUsageListBfBenchmarkModule(void);
void vRecordShowHeapUsage( char * CharString,char indexValueChar, portSHORT TestEntryFlag );
void vRecordTimerCountLED_NoCritical(MeasureValues *pMyMeasureValues, portSHORT TestEntryFlag );
 void vTurnOnRedLED(portCHAR LedCtrl);
void vTurnOnGreenLED(portCHAR LedCtrl);
void vTurnOnYellowLED(portCHAR LedCtrl);
portMEASURE_TIMER vGetTimerSnopShot(void);
void vMeasureToolInitAndBenchmarkUartCreate(char * CharString);
void vRecordTimerCountLED(MeasureValues *pMyMeasureValues, portSHORT TestEntryFlag );
void vRecordTimerCountLED_NoCritical(MeasureValues *pMyMeasureValues, portSHORT TestEntryFlag );
void vPrintMeasureResult( portMEASURE_RESULT MeasureResultScaled,char * CharString1,char indexValueChar);
void vSetupMeasureTimerLEDs(void);
void vBenchmark_ISR(void);
void vBenchmark_Throughput(void);
```