Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2009

MSc THESIS

Modular RT-Motion USB Software Framework

Widita Nugraha Budhysutanto

Abstract

Philips Applied Technologies has developed the RT-Motion USB platform as a compact distributed real-time motion control platform, but the platform can still be improved by developing a more advanced software framework. The goal of this thesis project is to design a modular software framework to complement the RT-Motion USB platform with extendability, flexibility, and configurability.
The design focuses on the extendability of the platform by developing foundation building blocks to integrate software extension modules and device drivers easily. The design emphasizes the principle of simplicity to ensure the lowest possible overhead and highest reliability. The firmware is modular, which allows each module to concentrate on its own area.
The implementation of the design has been tested and is proved to provide extendability, flexibility, and configurability while incurring low overhead. The improvement to the RT-Motion USB platform is expected to extend the applicability of the RT-Motion USB platform to a broader application range.

CE-MS-2009-28

PHILIPS
Applied Technologies

Mechatronics Department

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Modular RT-Motion USB Software Framework

Modular and Extendable Software Framework for Embedded Real-Time Motion Control Systems

---

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Widita Nugraha Budhysutanto
born in Surabaya, Indonesia

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Modular RT-Motion USB Software Framework

by Widita Nugraha Budhysutanto

Abstract

Philips Applied Technologies has developed the RT-Motion USB platform as a compact distributed real-time motion control platform, but the platform can still be improved by developing a more advanced software framework. The goal of this thesis project is to design a modular software framework to complement the RT-Motion USB platform with extendability, flexibility, and configurability.

The design focuses on the extendability of the platform by developing foundation building blocks to integrate software extension modules and device drivers easily. The design emphasizes the principle of simplicity to ensure the lowest possible overhead and highest reliability. The firmware is modular, which allows each module to concentrate on its own area.

The implementation of the design has been tested and is proved to provide extendability, flexibility, and configurability while incurring low overhead. The improvement to the RT-Motion USB platform is expected to extend the applicability of the RT-Motion USB platform to a broader application range.

Laboratory : Computer Engineering
Codenumber : CE-MS-2009-28

Committee Members :

Advisor:	B.H.H. Juurlink, CE, TU Delft
Chairperson:	K.G.W. Goossens, CE, TU Delft
Member:	A.J.C. van Gemund, ES, TU Delft
Member:	Jan F. Broenink, Embedded Control Systems, UTwente
Member:	Gerard Haagh, Mechatronics, Philips Applied Technologies
Member:	Sait Izmit, Mechatronics, Philips Applied Technologies

To my parents,
Budhy Sutanto and Lie Ay Hwa

Contents

List of Figures

List of Tables

1 Introduction

In this thesis we describe the design and implementation of a modular software framework for the RT-Motion USB platform developed by Philips Applied Technologies. Such framework is needed because the current firmware is developed in a static manner without modularity in mind, which is hard to be extended. The new modular software framework which is developed is intended to provide flexibility, extendability and configurability to the RT-Motion USB Platform.

This project had been carried out in cooperation with Philips Applied Technologies as a part of their *Home Robotics* project.

This chapter is organized as follow. In Section 1.1 a brief overview of the RT-Motion USB platform and its firmware is given. Section 1.2 describes the limitation of the current firmware and defines the requirements for the new modular and extendable software framework. Finally the organization of this thesis is described in Section 1.3.

1.1 RT-Motion USB

The RT-Motion USB is a modular, compact-size and low-cost USB based motion control platform designed by Philips Applied Technologies. It is equipped with a versatile ARM7 microcontroller and various input and output ports. Being based on the USB 2.0 High-speed bus, the RT-Motion USB enables an affordable distributed motion control platform without the need of conventional bulky parallel cable. The compact-sized board enables it to be easily deployed to various distributed motion-control environments where size matters. The RT-Motion USB platform was developed as a Master degree graduation project of a student from Technische Universiteit Delft in 2007 [11].

The ARM7 microcontroller which is chosen to be the core of the RT-Motion USB is an NXP LPC2888 microcontroller. This microcontroller has an ARM7TDMI core which runs at 60 MHz clock frequency and equipped with 8 KB of cache memory. Being a system-on-chip design, the microcontroller is also equipped with 64 KB of Static RAM, 1 MB of Flash ROM, a High-Speed USB 2.0 Device controller, and series of analog to digital and digital to analog converters. The powerful and versatile ARM7TDMI microcontroller also promises a decent computing power to perform some motion-control algorithms and perform basic signal processing tasks.

1.1.1 Software Framework Overview

RT-Motion USB board is designed as a USB device, which is intended to be connected to Windows/Linux based personal computer. The software framework which supports RT-Motion USB consist of the firmware in the board itself, the USB device driver for the Host PC, the User API for creating the motion control application in the Host PC

and finally the real-time motion control application. In this section the detail of each software framework component will be described in more detail.

1.1.2 Firmware

All pieces of software that power the RT-Motion USB board is contained in a set of code which is called firmware. The firmware controls all part of the board, from the smallest role like switching on or switching off a LED up to handling USB data exchange. The firmware is currently designed to control the RT-Motion USB board and provide USB based real-time data exchange between the board and the Host PC. The support of software based encoder counter algorithms has also been implemented into the current firmware.

The firmware is written in C language and compiled using the Keil RealView Microcontroller Development Kit. The image of the firmware can be uploaded to the on-chip flash memory by using a JTAG cable or using the USB cable. The ability to upload firmware by using a USB cable enables end-users to upgrade the firmware by themselves.

The main functions of the firmware are initializing the microcontroller, setting up the USB subsystem and then going to operating mode. During operation mode the firmware is doing its job by listening to two USB endpoints, one is used for controlling the board and the other one is used for the real-time data exchange with the Host PC.

If necessary, the developer can also modify the firmware to include an additional program, but some serious effort is needed. There is already one successful effort of modifying the firmware to execute local motion control algorithm to control the actuation of robot eyes. This firmware modification can operate stand alone without PC intervention while running 4 channel PID controllers at 2 KHz control frequency. The difficulties of modifying the firmware to include some additional program are covered in more detail in Section 1.2.1.

1.1.3 USB Device Driver

From the nature of USB specification, the functions of a USB device always depend on the USB host. This condition also applies to the RT-Motion USB. RT-Motion USB will need a USB Host to control the behavior of the RT-Motion USB board. The USB device driver of RT-Motion USB has been developed for Linux and Windows platforms. The USB device driver supports various Linux operating systems, including the ones equipped with real-time extensions like RTAI and Xenomai. RTAI and Xenomai is described in Section 2.4.

The USB device driver utilizes two USB Endpoints which are used for the real-time data channel and for the service message channel. The USB device driver specifies the transfer type of the USB connection as USB bulk transfer to be able to utilize the maximum bandwidth that is offered by USB. But the choice of using bulk transfer mode introduces one important restriction to the users. The user must ensure that the USB host controller that is used to connect the RT-Motion USB is not shared with other kinds of USB devices. This restriction is based on the characteristic of bulk transfer mode that it can achieve its best performance on an idle host. If the USB host controller is shared

with other devices, there might be the case that the other devices are also saturating the bus and automatically decrease the available bandwidth for the RT-Motion USB.

1.1.4 User API

In practice, currently the motion control algorithm is executed in the USB host PC, in a real-time motion control application built on top of a Linux system with some sort of a real-time extension. The real-time motion control application can communicate with the RT-Motion USB board by calling the functions included in the User API. The User API provides two group of operations, setting up board configuration and performing control data exchange. The board configurations which can be set up by using the User API are:

- Software Encoder Algorithm
  - Enable/disable software encoder counter
  - Setting encoder counting frequency
  - Choosing software encoder counting algorithm
- Analog/Digital I/O
  - Enable/disable A/D converters
  - Enable/disable D/A converters
  - Setting the Digital I/O Mask
  - Turning on/off onboard status LEDs
- Motor Amplifier
  - Enable/disable Amplifiers
  - Choosing amplifier decay mode
  - Choosing amplifier sleep mode
  - Selecting amplifier blank pin

1.1.5 Real-time Motion Control Application

The real-time motion control application is the user application which resides on the Host PC machine which can be in the form of a Linux application, a real-time Linux application or a Windows application. The real-time motion control application can communicate with one or more RT-Motion USB boards. This application communicates with the RT-Motion USB boards by calling some functions which have been defined in the User API.

After the data acquisition is done, the application can compute the actuation value according to the used control algorithm. The actuation value is then sent back to the RT-Motion USB boards by calling some functions in the User API. The data acquisition, processing and sending back to the RT-Motion USB boards must be done in a real-time manner. The frequency of the periodic loop can be adjusted to each application's requirements. This typical usage scenario for RT-Motion USB is shown in Figure 1.1.

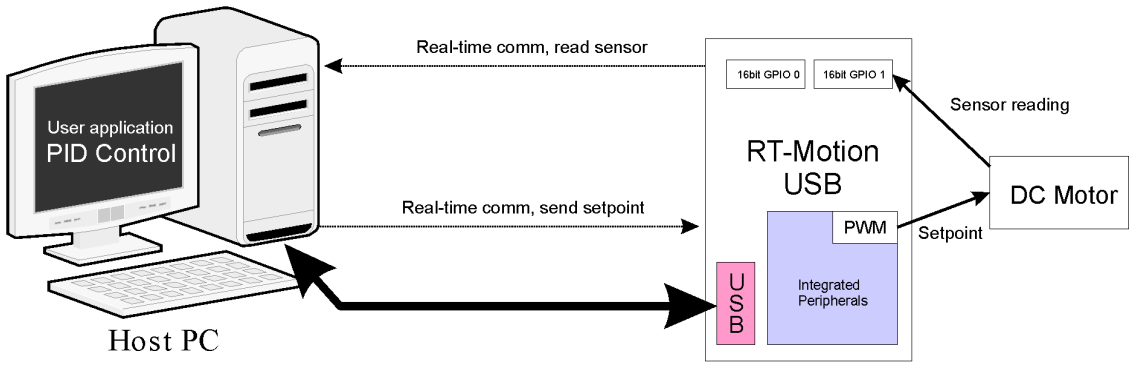

Figure 1.1: Typical Setup of RT-Motion USB

1.2 Problems Statement

This section covers the problems of the current system, the planned future system and how to achieve that future system. The formulated requirements document is delivered in appendix A.

1.2.1 Current

The current implementation of the firmware has already met the initial objective which is to make an affordable distributed real-time motion control platform. The platform has been deployed to many motion control applications and proved to be able to deliver good performance. But typically the full potential of the board is not yet utilized because it is usually used as a versatile distributed real-time I/O boards by primarily taking the advantage of the USB 2.0 high speed bus. The processing power of the LPC2888 microcontroller is still barely used because the standard firmware only focuses on the basic real-time message exchange operation.

The available processing power of the LPC2888 leads to the intention to write software to be executed in the firmware. But the firmware is built statically without the infrastructure to support extendibility. The static implementation is making it hard for the user to add some functions to the firmware or to modify some functions. To write extension software to the current firmware, the user must fully understand the firmware beforehand. Some of the important drawbacks present in the current firmware implementation are:

- To execute custom code locally the developer must modify the firmware manually, forcing the developer to study how the whole firmware works and integrate their custom code.
- When a developer needs to interface some hardware modules to be used with the RT-Motion USB boards, the developer has to implement the interfacing driver inside the firmware, which still does not have any well defined standard. And again this forces the developer to learn about the firmware implementation.

- The real-time data exchange in the current implementation is not done efficiently. All existing data fields are transmitted in the data exchange process even though that data is not being used. This approach is introducing a limiting factor to the extendibility of the system, because every time a new functionality is added, the developer must also modify the data exchange packet format accordingly.
- The housekeeping to make sure adding new functionalities will not harm the stability of the RT-Motion USB is quite difficult.
- The development of software extension modules and hardware extension modules requires the developer to prevent any resource conflict by manually keeping track of the usage of all on-chip resources and carefully assigning resources for new modules. This can be a very complicated process as the number of extension modules increases.

The main factors which limit the extendibility of the firmware are:

- The firmware does not have a basic framework which allows developer to write code on top of it.
- In-depth knowledge of the firmware's inner working is needed to start modifying the firmware.
- The developer has to take care of the usage of all onboard resources.
- The developer must make sure that the modification which is being made will not interfere with the real-time capability of the whole system.

1.2.2 Future

For achieving better utilization of all the features that RT-Motion USB potentially offers, some improvements must be made to the current software framework. The improvements of the software framework will include the improvement of the firmware itself, the User API and also the USB communication protocol. Because the current USB device driver has been proved to be able to provide a real-time communication channel for the current system, the USB device driver will be used as it is.

The new firmware will be equipped with building blocks which supports the developer to easily and rapidly develop software extension modules and device drivers to interface new hardware modules. The new software framework will have a more advanced real-time messaging protocol which supports configurable message formats. All improvements that is going to be implemented to the new software framework must be accompanied with a good configuration method to let the user customize RT-Motion USB according to the application's requirement.

1.2.3 How

Following are the steps needed to develop the new modular and extendable software framework for RT-Motion USB:

- Knowledge of the microcontroller being used, NXP LPC2888.
- Experience with the Keil Microcontroller Development Kit.
- Knowledge of the USB 2.0 high speed specifications.
- Formulation of robust communication protocol between PC and the USB board.
- Research on building the modular and extendable firmware.
- Research on the suitable real-time scheduler to be used.
- Research on how to manage the resources of the system.
- Research on the software extension modules and the integration to the firmware.

1.3 Thesis Organization

This thesis is organized as follows, Chapter 2 presents the background works related to the project, as an overview of the literatures that has been studied. Chapter 3 presents the design and implementation of the software framework which is the center of this thesis research. Chapter 4 presents the experiment results and discussions. Finally chapter 5 presents the conclusions and recommendations.

2 Background

This chapter covers the background of thesis project. The first part will cover the NXP LPC2888 which is used as the backbone of the system, and the Keil RealView MDK as the development environment. Some introduction to the USB 2.0 standard and the configuration used for realizing the real-time communication channel will also be covered in this chapter. And some ideas which might give useful insight for defining the communication protocol will be discussed. This communication protocol will be introduced to the new software framework for ensuring the efficient utilization of the communication channel.

2.1 NXP LPC2888 & SDK

This section discusses the microcontroller used in the RT-Motion USB and the development environment used to build the firmware.

The microcontroller which is used as the backbone of the RT-Motion USB is the NXP LPC2888 microcontroller. LPC2888 is a system-on-chip microcontroller which is based on an ARM7TDMI core. It includes a USB 2.0 High Speed device interface, an external memory interface that can interface to SDRAM and Flash, an MMC/SD memory card interface, A/D and D/A converters, and serial interfaces including UART, I^2C, and I^2S. Architectural enhancements like multi-channel DMA, processor cache, simultaneous operations on multiple internal buses, and flexible clock generation help to ensure that the LPC2888 can handle more demanding portable applications while requiring low power. The ARM7TDMI core inside the NXP LPC2888 can run at maximum clock frequency of 60 MHz. The microcontroller is equipped with 8 KB cache, 64 KB Static RAM and 1 MB Flash memory. In the implementation of the RT-Motion USB board, the external memory interface is not used, which limits this project to utilize only the on-chip memory.

ARM7TDMI The acronym ARM7TDMI specifies that the ARM7 core supports the ***T***humb 16-bit compressed instruction set, has on-chip ***D***ebug support that enables the processor to halt in response to a debug request, includes an advanced and faster ***M***ultiplier unit that can produce full 64-bit result and has an Embedded***I***CE hardware to support on-chip breakpoint and watchpoint. ARM7TDMI executes the ARMv4T instruction set architecture with Thumb extensions [1]. ARM7TDMI processors have been greatly used in mobile telephone handset applications where they are usually combined with a sophisticated DSP processor in a single chip solution. Here the ARM7TDMI has become the de-facto standard processor responsible for the control and user interface function the mobile telephone handsets [6].

The Thumb 16-bit compressed instruction set introduces a set of 16-bit long instructions which allows an almost double code density while giving almost the same performance as the standard 32-bit ARM instruction set. Despite being a 16-bit code, the Thumb code still uses the 32-bit registers of the ARM processor with some restrictions. Thus it can deliver performance which is more or less comparable to the 32-bit ARM instruction set. In order to achieve only 16-bit instruction length while still being able to perform all functionality, some adjustments must be made:

- No conditional instruction execution.
- Many Thumb instructions use a 2-operand format by using the same register for source and destination register.
- Fewer regular instruction formats because of the need of denser encoding.

One of the most sophisticated features of the Thumb Architecture Extension set is the ability to interwork seamlessly between the ARM and the Thumb code. This feature gives the developer the freedom to get the balanced code density and performance optimization.

Known Problems The NXP LPC2888 has a quite serious erratum regarding the execution of Thumb code from the on-chip Flash, which will cause a data abort exception if attempted. There is no known workaround for this problem, this erratum needs a serious attention when the developer wants to use the Thumb/ARM interworking. The erratum is documented in [25].

Keil RealView Microcontroller Development Kit The firmware for the NXP LPC2888 is written in C language and compiled using the *Keil RealView® Compilation Tools (RVCT)*. According to Keil, these development tools allow developer to write ARM applications in C or C++ while achieving the speed of assembly language [15]. Keil RVCT includes a complete toolkit for ARM development:

- The RealView C/C++ Compiler (armcc),
- The RealView Macro Assembler (armasm),
- The RealView Linker (armLink),
- The RealView Utilities (Librarian and FromELF).

Keil RVCT is part of the Keil Realview Microcontroller Development Kit (MDK), together with the μVision IDE and the Keil RTX Real-Time Kernel.

2.2 USB 2.0 as Real-time Communication Channel

USB 2.0 High Speed The USB architecture is defined as a master-slave protocol where a host controller acts as a master and communicates with many client devices

which act as slaves. The communication must always be initiated by the master, thus the slaves can only reply to the master's requests.

According to the USB 2.0 specification [3], three bus speeds are supported: low speed at 1.5 Mbit/sec, full speed at 12 Mbit/sec, and high speed at 480 Mbit/sec. The high speed standard is introduced with the release of USB 2.0 specification. The bus speed actually describes the rate of data transfer in the bus which is shared by all peripherals connected to it, so the rate of data transfer that can be expected by each device might be less than the bus speed. The theoretical maximum rate for a single data transfer is about 53.248 Megabytes/sec at high speed and around 1.2 Megabytes/sec at full speed. The low speed only offers around 0.8 Kilobytes/sec [2].

Communication Types The communication types in USB can be divided into two groups, the *enumeration communication* and the *application communication*. The enumeration communication is all form of communication that is needed for a host to enumerate and configure a USB device which has just been plugged in. During this enumeration process the operating system is recognizing what device is being plugged in, and tries to find the suitable drivers to control that device. On the other hand the application communication is all form of communication between the Host PC and the USB device that is needed for the device to perform its function and deliver its result, the communication is done between the user application and the USB device by calling operating system functions.

Endpoints An endpoint is a block of data memory in the USB device that acts as a buffer that stores multiple bytes of data which are ready to be sent or data which has just been received by the device. All traffic on a USB bus is from the USB host to a device endpoint or from the device endpoint to the USB host. An endpoint address is a combination of an endpoint number which ranges from 0 to 15 and the direction of the endpoint from the USB host's point of view, which is IN or OUT.

A device must have at least one control endpoint, called endpoint 0. This endpoint is used as a bidirectional endpoint named endpoint 0 IN and endpoint 0 OUT. A device can have maximum 15 other endpoints numbers which each can support both IN and OUT endpoint addresses that makes total 30 endpoints addresses. An association of a device's endpoint to the host controller's software which is needed to perform a transfer is called a USB pipe.

Transfer Types To accommodate a broad range of devices which might have different transfer rate, response time and error correcting requirements, USB defines four types of data transfer mode:

1. Control
   The control transfer type is a required transfer type that must be implemented by every USB device. It is usually used by the host to identify the USB device and set the device configuration.

2. Bulk
   The bulk transfer type has the maximum transfer rate potential, theoretically

around 53.248 MB/sec. But that potential can only be achieved when the bus is idle; when the bus is very busy there is a possibility that the transfer can be delayed. USB guarantees neither the delivery rate nor the latency, but it will try up to three times in case of failed delivery. The real world performance of bulk transfer at the time of the writing of [2] is up to around 35 MB/sec.

3. Interrupt
   The interrupt transfer type is usually used for mouse and keyboard devices. All interrupt and isochronous transfers (see below) have a combined reserved bandwidth 80% of the USB bandwidth for USB high speed and 90% for USB low speed and full speed. The latency or maximum time between transfers is guaranteed. The name interrupt transfer can be deceiving because this transfer still requires the host to poll the device to check if there is a pending transfer.

4. Isochronous
   Isochronous transfer is typically used for streaming applications, for instance, encoded voice and music to be played in real-time. There is no error correction mechanism implemented. An isochronous transfer can make sure that a transfer is possible to be performed quickly even in a busy bus. Here the latency of the message is guaranteed, but the successful delivery of a message is not guaranteed.

All USB transfer types have error correction mechanisms implemented, except the isochronous transfer type.

Real-time communication via USB 2.0 High Speed RT-Motion USB provides a real-time communication channel which is based on USB 2.0 High Speed at 480Mbit/s. The USB transfer type is set to bulk transfer mode to be able to use all available bandwidth that USB 2.0 High Speed could offer, but the nature of USB bulk transfers mode does not offer any data delivery timing guarantee. To make sure that the RT-Motion USB boards achieve the best performance there is a practical limitation that needs to be introduced, the RT-Motion USB boards should be connected to a dedicated USB host which is not shared with any other kinds of USB devices utilizing other transfer types. USB bulk transfer has an error checking mechanism already implemented. Experience also shows that bulk mode is the communication mode of choice for embedded devices [18].

USB Debugging To analyze what is going on in a USB bus to debug some low-level problems, there is a kind of USB tool called *USB bus analyzer*. There are many vendors offering USB bus analyzers, such as *Ellisys USB Explorer 200* [5] and *Total Phase Beagle USB 480 Protocol Analyzer* [20], in the price range of 1,000-2,000 EUR. The detailed pricing information and the comparison between both solutions can be found in [21]. USB bus analyzers are typically built as a set of hardware and software, the hardware is connected between the USB host and the USB device so the software can record all data, electrical states and control information transmitted on the bus without affecting the communication or device behavior. The typical hardware setup can be seen in Figure 2.1. The software is installed on the host PC and is responsible for the capturing process.

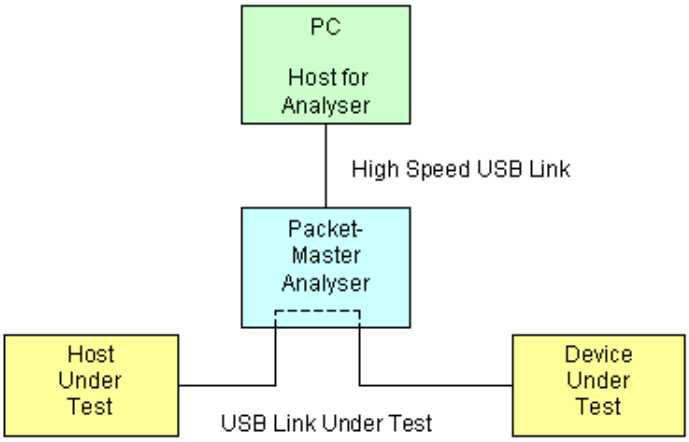

Figure 2.1: USB Bus Analyzers Hardware Setup

An alternative to USB bus analyzer tools is the Linux kernel's *soft USB tracer*, *usbmon*. This tool can also capture the traffic on the USB bus. To use this tool the developer must ensure that *debugfs* (CONFIG_DEBUG_FS) and *usbmon* (CONFIG_USB_MON) support has been enabled in the Linux kernel. There is also a website dedicated for ideas on USB testing [23].

2.3 Communication Protocol between RT-Motion USB and Host PC

The development of the communication protocol is also important to realize real-time communication between all the USB boards and the Host PC. The overhead of adding more USB boards to the Host PC also needs attention in the development process. The usage of the communication channel can be optimized by employing a custom tailored packet for each kind of configuration which is being used, which will need some sophisticated protocol to support that.

The protocol must have a good efficiency to meet the performance requirements of the real-time motion control system and it should be scalable in terms of accommodating various data exchange requirements which may be different per motion control environment.

The communication protocol now becomes important because of the enormous combination possibility of data which can be exchanged between the USB boards and the Host PC. The packet format must be dynamically formatted depending on the need of the RT-Motion USB boards, but still delivers a good real-time performance. To get more insight about some packets mapping implementation some basic principle of the Fieldbus memory management unit from EtherCAT and process data objects from CANopen are covered in the next section.

2.3.1 Fieldbus MMU from EtherCAT

EtherCAT (Ethernet for Control Automation Technology) [7] is an automation network which works on an Ethernet network but introduces a "processing on the fly" approach. This approach enables EtherCAT to minimize delays and jitters in the communication because data can be extracted and inserted to the Ethernet frame on the fly. From the master node's point of view, the whole EtherCAT system is seen as a large distributed memory that can be read and written without restriction.

The *Fieldbus Memory Management Unit (FMMU)* is a mechanism used for mapping process data from the logical process data image on the EtherCAT Fieldbus to the physical memory address of the local device. The FMMU can even map the process data bit-wise, meaning that a data item which is represented as a single bit can be inserted anywhere in the packet's logical address space. FMMU is implemented in an EtherCAT Slave Controller.

Every EtherCAT slave has its own FMMU mapping table, the table is responsible for keeping track of the mapping of every logical address which came inside a packet to a physical memory address inside the EtherCAT slave device. The table is set at the initialization phase of the EtherCAT networks. Just by checking the mapping table, every slave node can retrieve information from the packet and insert information to the packet as instructed by the master node, this process is done as the packet goes through the slave nodes.

2.3.2 Process Data Objects from CANopen

CANopen [10] is a higher layer protocol for Controller Area Network (CAN) based networks that describes the data exchange mechanism. CANopen defines both the communication profile and the device profile. The core of any CANopen node is a lookup table with a 16-bit index and 8-bit sub-index which is called Object Dictionary [9]. The object dictionary stores all process and configuration data as entries in predefined locations. The messages which are used to read or write any node's object dictionary entries are called Service Data Objects (SDO).

Process Data Objects (PDO) is the message format that is used to transfer process data in the object dictionary. With the help of PDO mapping inside the object dictionary entries, any data which is listed in the object dictionary entry can be mapped to a data inside a PDO message and transferred to other nodes. The maximum length of data that can be transmitted by a PDO message is 8 bytes. If there is a need to transfer more than 8 bytes, the data transfer will be automatically fragmented into multiple PDO message.

2.4 Real-Time Functionality Support for Linux on Host PC

The RT-Motion USB software framework consists of the firmware on the RT-Motion USB board itself and the software package on the Host PC which controls it. In order to

make a real-time motion control application, both sides must be developed in a real-time manner. On the host PC there are quite a number of real-time Linux kernel patches options which can be used.

RTAI RTAI (Real-Time Application Interface) [26] is a real-time Linux kernel extension which allows real-time tasks to run alongside standard Linux tasks, RTAI itself is not an RTOS. RTAI can provide deterministic response to interrupts by using the *Adeos virtualization layer*. RTAI provides a native API with lots of services to support the real-time application development.

Xenomai Xenomai [27] is a real-time development framework which runs in cooperation with the Linux kernel to provide hard real-time support to user-space applications. Xenomai is built in a layered manner, including a hardware abstraction layer, a real-time nucleus and various kinds of skins. The layered approach and support of RTOS skins give Xenomai better extensibility than RTAI.

RT-Preempt Patch Unlike the previous attempt of realizing a real-time operating systems by creating an extension to the Linux kernel, Ingo Molnar and a small group of developers [17] tried to make a true real-time Linux operating system by modifying the Linux kernel. The modification is combined in a patch set which is called the RT-Preempt patch set. RT-Preempt patch converts conventional Linux kernel into a fully preemptible kernel and thus gains hard real-time capabilities. The real-time patch includes the replacement of spinlocks with preemptable mutexes which enables involuntary preemption anywhere within the kernel except for the protected areas [8].

In the current RT-Motion USB application in Philips Applied Technologies, Xenomai is chosen as the Linux real-time extension for hosting the real-time motion control application. The RT-Preempt patch is not chosen because the patch is still in a development stage. But as the development of all solution advances, RT-Motion USB software framework can be adapted to run on the most promising platform.

2.5 Scheduler

One of the most important aspects of a motion control platform is the timing characteristic. To get the best performance from a motion control platform, the system must be a real-time system. Real-time system is a system whose result is not only evaluated by the functional result but also the delivery time of the result. There are two kinds of real-time systems:

- Soft Real Time
  A soft real-time system is a system in which its operations have deadlines, but when a deadline is missed, the effect is not fatal to the system. An example of a soft real-time system is a live television broadcasting system, where the system must process the video signal and broadcast the signal via various broadcasting medium in real-time. When the video processing system encountered a glitch and

produced a delay which is larger than an acceptable limit, some video frames can be skipped without leaving a catastrophic failure. At most the user's experience quality is decreased.

- Hard Real Time
  When a deadline is missed, the result to the system can be catastrophic. A single deadline miss can have a fatal effect to the system, so the system must make sure that all deadlines can be met on-time. An example of a hard real-time system is the airbag controller in a car; in which if a deadline is missed the passengers' safety can be in danger.

The motion-control application can be in both categories. Sometimes it can be a soft real-time system, but for some cases it can also be a hard real-time system. By allowing the integration of software extension modules, we introduce additional complexity to the system, the system can no longer guarantee that with all software extension modules integrated, the system can still perform within the required timing constraints. In order to fix this problem we will need a real-time scheduler which is responsible for maintaining the real-time capability of the overall system.

Real-time schedulers can be divided into two broad classes of scheduler [22, pp. 246]:

- Non-preemptive multitasking
  This kind of scheduler lets a task execute until it voluntarily returns the CPU to the scheduler. This kind of scheduler is simple, predictable, reliable and safe.

- Preemptive multitasking
  This kind of scheduler can assign different priority levels to each task and a task with higher priority can always preempt any task with lower priority. This kind of scheduler are generally considered less predictable, less reliable than non-preemptive schedulers.

A real-time scheduler can be obtained by taking one off the shelf real-time operating system solutions or writing a real-time scheduler from scratch. The option of building our own real-time scheduler from scratch can be a more reliable solution for a simple application, because we can track our own code during the software lifetime and automatically increase the software maintainability.

But there is also a drawback of this approach, especially when the requirements of the system is getting more complex, the scheduler code will also be more complex and need a lot of development and maintenance effort. For instance, when the system eventually needs a preemptive scheduler, building your own preemptive scheduler can be a non efficient approach, considering the highly complex structure it introduces. The preemptive real-time scheduler will eventually need to be equipped with some semaphore management and message passing mechanism to deal with critical section problems which may occur.

When the system only needs a cooperative multitasking scheduler, building a real-time scheduler from scratch will be a good approach. The scheduler can be built customized to the system's requirement while trying to achieve the lowest scheduler overhead.

The kinds of the off the shelf real-time operating systems solutions that have been considered during this thesis research are:

1. Embedded Linux
   It is a ported version of a full fledge Linux, with features similar to x86 Linux platform, but requires a great amount of memory resources, RAM and Flash storage. It is not suitable for our needs because we are focusing on the smallest possible footprint.

2. Proprietary Embedded RTOS
   There are many kinds of proprietary real-time operating systems which have been used widely, like Wind River VxWorks [24] and MontaVista Linux 6 [19]. These kinds of RTOSes have found its place in many Wireless Router applications. These RTOSes offer complete facilities but require a relatively large footprint.

3. Mini Kernel
   Mini kernels are small operating systems which only provide some basic scheduler functions and message passing features. Examples of this kind of RTOSes are the FreeRTOS, Keil RTX Real-Time Kernel, and many others. A comparison between various mini kernel off the shelf RTOSes is given in Appendix B.

Considering the limited resources that RT-Motion USB has, which are 1 MB of Flash ROM and 64 KB of Static RAM, the chosen RTOS solution must be the one based on the mini kernel solution.

2.5.1 Keil RTX Real-Time Kernel

The Keil RTX Real-Time Kernel is a real-time kernel for ARM7, ARM9, and Cortex-M3 devices [13]. The structure of the RTX kernel can be observed in Figure 2.2.

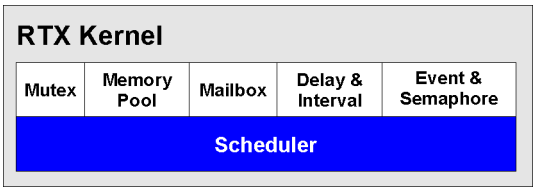

Figure 2.2: Keil RTX Real-Time Kernel Diagram

The real-time kernel supports a round-robin scheduler, a cooperative scheduler and a preemptive scheduler. There are also some inter-process communication mechanisms and mutex/semaphore implementation to support preemptive scheduling.

Experiments During the experiment part of this thesis, some experiments are made on the process of running Keil RTX Real-Time Kernel on the LPC2888 based RT-Motion USB board. The steps taken in the experiment include the porting of the RTX kernel configuration file, some troubleshooting with the platform's known problems, and taking some measurement to get an idea about the kernel's performance.

The required modifications to be made to the RTX configuration file include the adjustment of the interrupt handling mechanism and implementing the detail of hardware timer setup for the target microprocessor.

The experiment stumbled into an issue that raised an exception to the microprocessor, a Prefetch Error exception, which was actually caused by the fact that the RTX kernel code was by default compiled as Thumb Code and the NXP LPC2888 has a known problem, a hardware erratum which restricts the execution of Thumb Code from the flash memory. This known problem is described in Section 2.1.

The experiment also focused on the performance of the RTX kernel, the performance measurement includes the task switching time and the overhead which is caused by the scheduler. The timing measurement result can be observed in Table 2.1. The timing unit used in the measurement is microseconds.

Table 2.1: Keil RTX Timing Specifications

nFunction	ARM7 @ 60MHz	LPC2888 @ 60MHz		
		min	max	avg
Initialize system (os_sys_init)	46.2	62.2		
Create task, no task switch	17.0	22.7		
Task switch (by os_tsk_pass)	6.6	8.67	9.51	25.37

In general the outcome of the experiment showed that the overhead of the scheduler is too big, in some cases the execution of task can be exposed to around 19 microseconds overhead from the scheduler's interrupt service routine code. And the experiment can not reproduce the task switching time as stated by Keil. The task switching time is around 25-30% slower than the reference published by Keil in [12].

After seeing the result of this timing experiment we decided not to use the Keil RTX Real-Time Kernel in our future software framework due to the high overhead which is introduced. Keil RTX Real-Time Kernel also does not come with the source code, which means that we cannot modify the kernel at all to simplify or optimize it.

2.6 Resource Management

The RT-Motion USB has many kind of resources which can be shared by all software and hardware modules which are installed. To make sure that all installed modules can function properly, all their resource requirements must always be satisfiable by the system.

There are two kinds of shareable resources in the system:

- Exclusive resource
  During the whole system runtime, the resource usage is only granted to a software module exclusively, the resource allocation is done during the initialization phase of the system. The examples of this kind of resources are the physical GPIO pin on the board or the physical ADC channel. The usage of this kind of resources are known before the code execution, therefore it can be assigned beforehand. The

resource manager can have a resource allocation table which keeps track which resources are free and which resources are being used by which modules.

- Non-exclusive resource
  If the usage of a resource is not exclusive to a module, there can be the case when two or more modules try to access the resource at the same time. The example of this kind of resources is a memory variable in the system. Some modules can have the write access to a variable while the other can have a read access to the same variable, the usage of the access must be managed to avoid conflicts. The access to a shared resource can be seen as a critical section if the system utilizes a preemptive scheduler, which can be guarded by some RTOS facilities like semaphore.

2.7 Modular and Extendable Firmware

The main goal of this project is to make the RT-Motion USB more extendable and to give more flexibility to the platform. To achieve that goal, we have to enable some way to utilize all the potential that the hardware offers. One of the important features which needs to be implemented is the ability of the firmware to integrate firmware extension modules easily. The firmware extension modules can be categorized into two kinds, software extension module and device driver.

Software extension module can be any kind of code which can be executed on the ARM7 processor to support the motion control process. It is designed to be able to communicate with the host PC via a real-time messaging facility. The kinds of software extension modules are limitless; it can be a simple code to interface some simple sensors which is connected to any onboard port, a PID controller, or a filter to improve the data acquisition.

A Device driver is similar to software extension modules but the purpose of a device driver is to control certain hardware functionality easily. This is used to enable the developer to make a new hardware extension module and make the necessary software to control that particular hardware extension module. The device driver contains the code that is used to control a device and to provide an interface for the software extension modules developer.

In order to enable the easy integration of firmware extension modules, the whole firmware must be redesigned with modularity in mind. The firmware will be designed by specifying some firmware building blocks to form a set of modules which cooperate with each others to provide overall functions. The firmware building blocks will consist of a configuration manager, a resource manager, a software extension modules manager and a communication manager. More detailed information about the requirements which have been formulated for each building blocks can be seen in Appendix A.

This section will cover the things that might reveal the potential offered by enabling the software floating point emulation support in the software extension modules and also study about the integration of software extension modules.

2.7.1 Soft Floating Point

ARM7TDMI processor core does not have a physical floating-point unit for performing floating point calculation and thus there is no floating-point instruction set available. When floating-point calculations are needed there are two alternatives, the first one is by installing a VFP Coprocessor next to the ARM7TDMI processor core and the second one is using the software floating-point library which is provided by ARM. The compiler will call a set of procedures which is contained in the software floating-point library, *fplib* [16] to perform floating-point arithmetic.

Since the focus of this project is extending the software framework without doing any hardware modifications, *fplib* will be investigated. *fplib* is available as a part of the standard distribution of the RealView Development Suite C libraries, which is included in the Keil Microcontroller Development Kit that is used.

The software floating point library can be used by the Keil compiler when the compiler is called with the command-line option *–softvfp*. Using software based floating point arithmetic, the performance of the floating-point operations is not as fast as using a floating-point coprocessor. The performance of the software floating library is measured and investigated to find the possibility to use it for performing some basic motion control algorithms. The result of the measurements can be observed in Table 2.2. The time unit of the measurements is in microseconds.

Experiment Methodology The experiment is done by making a series of floating point numbers to be processed using the floating point library. These numbers are used as the operand of the floating point operations. The timing measurement is performed by taking the value of a free running timer right before the execution of the floating point operation and after the operation is finished. The values shown here are an average of 360 different operand values and each test is repeated 10 times.

Table 2.2: Software Floating Point Library Performance Analysis

Operation	Float			Double		
	Minimum	Average	Maximum	Minimum	Average	Maximum
IntToFP	0.72	1.38	1.38	0.78	1.35	1.35
FPToInt	1.02	1.5	1.47	1.53	2.05	2.27
Add	1.33	1.65	2.48	1.9	2.57	4.03
Sub	1.35	1.77	2.37	1.82	2.68	5.05
Mul	1.22	1.6	1.6	1.7	3.92	4.03
Div	1.28	2.98	3	2	10.63	10.87
Sqrt	2.75	13.32	13.35	3.38	31.17	31.95
Cos	1.32	57.33	67.13	4.85	77.92	92.27
Sin	1.38	57.47	67.27	4.27	78.9	91.58
Tan	1.28	59.87	69.53	5.22	150.3	183.4

The performance of the software floating point library can be considered good enough. Motion control algorithm mostly uses addition, substraction, multiplication and division operations which showed to have good performance. The software floating point library

will be used in the RT-Motion USB platform for enabling a floating point control algorithm.

3 Design and Implementation

This chapter describes the overall system architecture and the design of the modular and extendable software framework for the RT-Motion USB. The software framework consists of the firmware, the firmware extension modules and the host PC supporting framework. It is organized as follows: in Section 3.1 the overall system architecture of the new system is described. Next, the structure of the modular RT-Motion USB software framework is described in Section 3.2. In Section 3.3 a description of the design of the firmware is given. Followed by the discussion of the firmware extension modules in Section 3.4. Finally, the supporting framework which runs on the Host PC is covered in Section 3.5.

3.1 System Architecture

The section describes the system architecture of the RT-Motion USB in general. The operating mode of the RT-Motion USB is defined in Section 3.1.1. In section 3.1.2 the communication model of RT-Motion USB with the Host PC is described. In Section 3.1.3 a description of the communication model of RT-Motion USB with the hardware extension modules is given.

3.1.1 Operating modes

This section describes the usage scenario of the RT-Motion USB boards in motion control applications. The RT-Motion USB board was originally designed to work as a distributed real-time I/O board. By utilizing USB 2.0 High Speed connection, it is possible to plug many RT-Motion USB boards to one Host PC and distribute the boards in a motion control environment within the maximum achievable range of USB bus. When the RT Motion USB boards are installed in this configuration, they are operating in *slave mode* configuration.

Considering that the RT-Motion USB board has great I/O peripherals and the relatively high computing power, it is also suitable to be used in a single node motion control environment. When the RT-Motion USB board is configured to run without communicating with a PC, it is called the *stand-alone* mode.

The following sections describe the slave mode and the stand-alone mode configuration in more detail.

Slave Mode The slave mode is a configuration where the RT-Motion USB board is used to support the motion control software executed on the Host PC. The motion control software on the Host PC communicates with the RT-Motion USB board by exchanging *real-time data objects (RTDO)*. A more detailed information about an RTDO is described

in Section 3.1.2.5. When necessary, the application in the Host PC can communicate with more than one RT-Motion USB boards. This enable the implementation of a multiple input multiple output controller.

Figure 3.1 shows an example system configuration which consists of two RT-Motion USB boards connected to a Host PC working in a slave mode. The figure also shows some hardware extension modules connected to the RT-Motion USB boards. These hardware extension modules are described in Section 3.1.3.

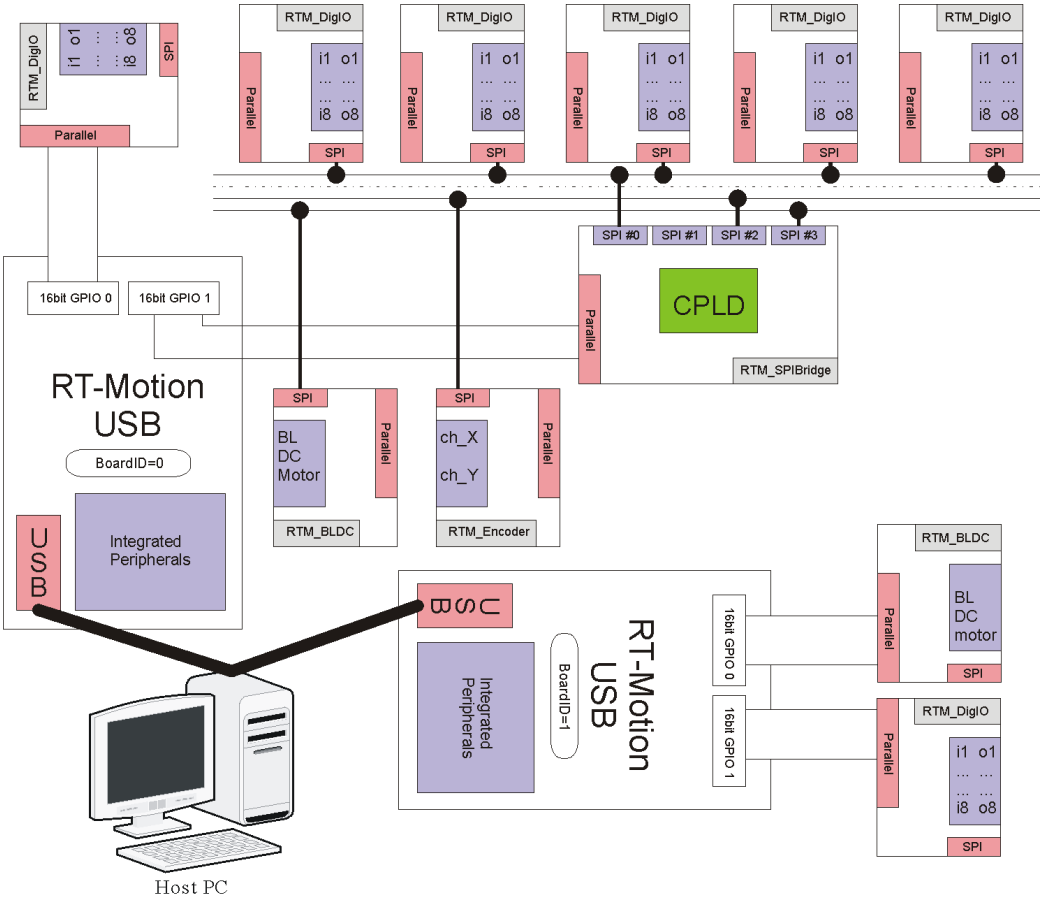

Figure 3.1: RT-Motion USB Slave Configuration

Stand-Alone Mode The Stand-Alone mode is a configuration where one RT-Motion USB is performing a specific motion control task without the need of a Host PC intervention. The Host PC is only needed to configure the RT-Motion USB board for the first time and to instruct the RT-Motion USB board to save the defined configurations.

The RT-Motion USB can configure itself from the pre-programmed configuration and start operating without establishing any USB connection with the Host PC.

Figure 3.2 shows an example of one RT-Motion USB board in controlling an electric motor in a stand-alone mode.

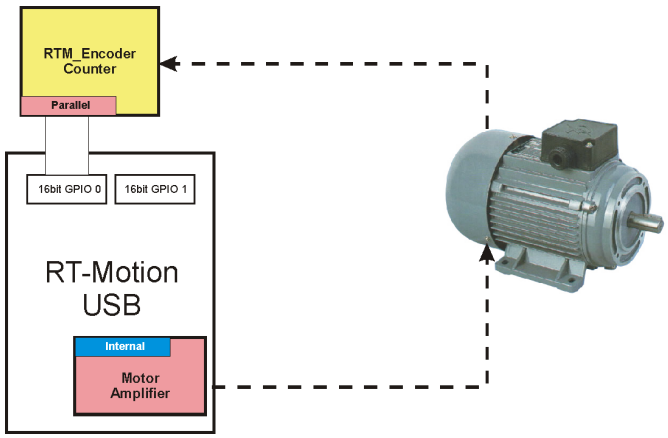

Figure 3.2: RT-Motion USB Stand-Alone Configuration

3.1.2 Communication Model of RT-Motion USB with Host PC

The RT-Motion USB board is communicating with the Host PC via a USB bus. The USB communication between the Host PC and the RT-Motion USB is designed to utilize two channels, a *real-time channel* and a *service channel*. By splitting the USB communication into two different channels, the processing mechanism can be optimized for each channel. The service channel and real-time channel by nature have different requirements. A service channel is mostly used for sending configuration to the RT-Motion USB board during the configuration phase, while a real-time channel is used during a real-time operation of the board. The real-time channel is given a higher processing priority to achieve a real-time communication performance. Figure 3.3 shows the diagram of the USB communication model in the system.

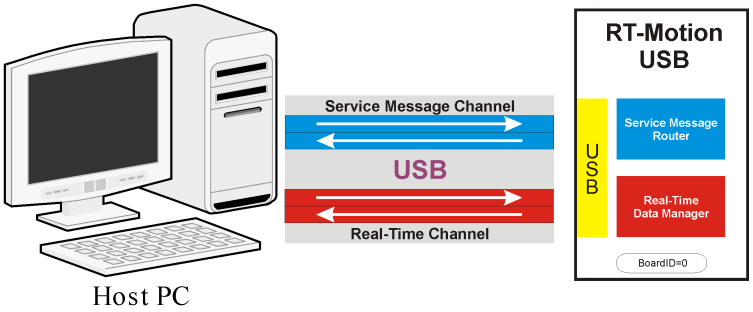

Figure 3.3: USB Communication Model

3.1.2.1 Service Channel

Even though a service channel is mostly used for sending and receiving configuration parameters during the configuration phase, it is also available in the real-time running mode for sending urgent commands to the RT-Motion USB board. The service channel is utilizing an endpoint which is configured as a bulk transfer type. The processing of the service channel is handled by a *Service Message Router (SMR)*.

An SMR is designed to be the central manager of the message exchange in the service channel. Every module which intends to have an active service channel communication registers its service message handler to the SMR. The role of the SMR here is to parse the incoming *service message*, identify the destined receiver and route the message to the destination module. The message interpretation and replying is then left to the destination module's freedom. This routing process is implemented inside the SMR in the form of a high priority sporadic task. This way the USB interrupt service routine execution time is kept at minimal. The SMR is described in more detail in Section 3.3.4.

3.1.2.2 Service Message

The type of message which is exchanged on the service channel is called a service message. A service message can be addressed to any module in the firmware side; each module will process that message on their own way. Because a service message is designed to transport configuration messages, the direction of the message is always from the Host PC to the firmware module. The recipient of a service message, by definition, has an obligation to send a reply to the service message.

The service channel is implemented to support the handling of a single message at a time, meaning that a service message must be replied before the host PC is allowed to send another message. If a new service message is received by the RT-Motion USB board when there is a service message in process, the new service message will be ignored. The single service message design allows a simpler implementation of the SMR while keeping the overhead as low as possible.

Since the system only support a single service message, only one buffer is needed for storing the service message. This buffer can be used for both buffering the incoming and outgoing service message. This buffer is allocated and maintained by the SMR. The pointer of the buffer is passed to a USB Communication Layer during SMR's initialization. The USB Communication Layer is described in Section 3.3.3.

A service message has a header which practically can be divided in two different parts. The first part of the header is the information which is used by the SMR to identify the recipient of that message. The second part is the information which is used by the recipient itself, which includes the message identifier and the length of the payload. The size of the service message is flexible with a maximum size of 512 bytes, defined as the maximum message size for bulk transfer mode by USB specification [3, pp. 53]. Figure 3.4 shows the structure of a service message.

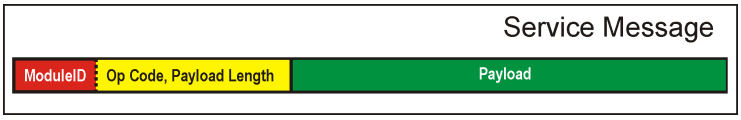

Figure 3.4: Structure of a Service Message

3.1.2.3 Real-Time Channel

The real-time channel utilizes an endpoint which is configured as bulk transfer type. The bulk transfer type is chosen because it can fully utilize the available bandwidth of the USB 2.0 High Speed (480 Megabits per second) when the USB bus is not shared with devices utilizing other transfer types. The real-time channel will be used to exchange *real-time messages* only. The processing of the real-time channel traffic is handled by the Real-Time Data Manager (RTDM).

An RTDM is the firmware module responsible for receiving and transmitting messages on the real-time channel. The design of the RTDM is described in Section 3.3.5.

3.1.2.4 Real-Time Message

A Real-Time Message (RT Message) is a form of message exchanged between the USB host and the RT-Motion USB in a real-time manner. A real-time message can have a variable size, but the maximum message size including the header is 512 bytes. This maximum size is tied to the maximum size possible in the bulk transfer mode. The payload of a real-time message is a collection of real-time data objects (RTDOs). RTDOs which need to be exchanged together should be grouped into the same real-time message where applicable. A more detailed information about RTDO is described in Section 3.1.2.5. An example of a real-time message which contains RTDOs is shown in Figure 3.5.

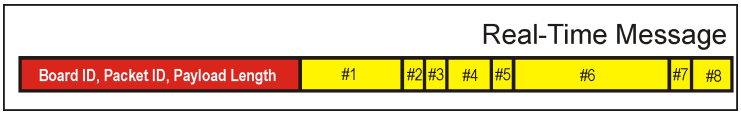

Figure 3.5: Real-Time Message containing RTDOs

According to the direction of the message, real-time messages can be grouped into two categories, incoming real-time messages and outgoing real-time messages. The framework supports multiple types of RT messages for each direction. Every type of a real-time message is identified by a unique *PacketID*.

The incoming real-time message is a real-time message type which is sent by the Host PC to the RT-Motion USB Board. Incoming real-time messages can be used to transport multiple RTDOs at the same time or to specify the type of outgoing real-time message that is expected by the host PC on the next request.

The outgoing real-time message is a real-time message type which is sent from the

RT-Motion USB board to the USB Host on request. An RTDM supports multiple types of outgoing real-time message, but the USB device controller requires that the internal FIFO buffer is filled with an outgoing real-time message when the host PC issues a read request. To provide the host PC with the correct message type as expected, RTDM must always be notified what is the type of outgoing real-time message expected by the host PC on the next request. The notification can be received by the RTDM from the Host PC via a special formed incoming real-time message or from any software extension module (SWEM) via a function call.

3.1.2.5 Real-Time Data Object

A real-time data object (RTDO) is a data object that can be placed in a real-time message to be transmitted either from PC to USB or from USB to PC. The size of the data object is customizable, ranging from 1 to 500 bytes. One real-time message can contain many real-time data objects together. Data objects that have the same delivery schedule should be placed inside the same real-time message.

An RTDO is used to transport information between the host PC and the RT-Motion USB board. The RTDO is placed inside a real-time message during the communication configuration state. One RTDO can only be placed into one real-time message, either the incoming real-time message or the outgoing one. The value of an RTDO can be read by any SWEM and can be updated by a SWEM.

3.1.3 Communication Model of RT-Motion USB with Hardware Extension Modules

The project of transforming the RT-Motion USB board into a more extendable and flexible platform is not only pursued from the software point of view but also from the hardware point of view. A paralel research is being conducted to make the RT-Motion USB board easily support hardware extension modules. This research is trying to define a hardware extension module development standards.

The hardware extension modules can be installed to the RT-Motion USB board with the following attachment mechanism:

- Two 16-bits GPIO parallel ports
  Up to two hardware extension modules can be attached to the onboard 16-bits GPIO parallel ports. The amount of available ports automatically limits the number of modules that can be attached to the RT-Motion USB board.

- SPI Bus
  In order to support more than two hardware extension modules simultaneously, a support for SPI bus is added to the RT-Motion USB. The SPI bus is supported by connecting a specially designed extension modules which functions as an SPI Bridge to the first 16-bits GPIO parallel port. The SPI Bridge module supports a maximum of 8 SPI devices.

  After a hardware extension module is installed, a specially developed device driver would allow the user of that module to communicate with the physical module. The design of the device driver is described in Section 3.4.2

3.2 Structure of Modular RT-Motion Software Framework

In this section we describe the structure of the Modular RT-Motion Software Framework designed in this thesis project. The software framework can be splitted into three main parts.

- Modular RT-Motion USB Firmware
  The Modular RT-Motion USB Firmware is the core of the framework. Most of the works in this thesis project is focused on the design of the modular firmware. The design of Modular RT-Motion USB Firmware is covered in more detail in Section 3.3

- Firmware Extension Modules
  The firmware extension modules are parts of the framework which are actually the result of the implementation of modular firmware design. The new firmware design allows the execution of extension modules such as the Software Extension Module (SWEM) and the Device Driver. Section 3.4 covers the design details of these firmware extension modules.

- Host PC Supporting Framework
  Because RT-Motion USB is a USB device, a software framework is also needed to be implemented as the supporting framework on the Host PC. The design of the Host PC supporting framework is described in Section 3.5

3.3 Modular RT-Motion USB Firmware

The Modular RT-Motion USB Software Framework consists of software running on the USB Device and the USB Host. This section focuses on the design of the Modular RT-Motion USB firmware.

As described in Section 1.2.1, the current firmware of RT-Motion USB is implemented as a static firmware without modularity. A static firmware is not always bad because it can provide a stable and reliable system for the intended purpose, but a great amount of effort will be needed to extend the firmware while maintaining the same level of reliability.

The fundamental firmware modules will be described in detail, followed by dependency of those modules. Then the state of the firmware, the initialization modes, the execution context, and the memory map of the firmware will be described in detail in this section.

3.3.1 Fundamental Firmware Modules

The firmware, consisted of 11 fundamental components, is designed in a modular manner. The modular firmware approach is taken because it will result in a more structured design. Basically each firmware module will have a responsibility for one specific region of the firmware. A well structured firmware design is important to realize a reliable complex firmware. The fundamental firmware components are:

- Hardware Abstraction Layer
  A hardware abstraction layer is a set of system defines, macros and functions which is used to access the hardware directly.

- USB Communication Layer
  The USB communication layer is the firmware module which handles the USB communication by controlling the USB device controller.

- Service Message Router
  The Service message router is the firmware module which handles the service channel part of the USB communication system. It handles the exchange of service messages between the host PC and the firmware modules.

- Real-Time Data Manager
  A real-time data manager handles the reception and transmission of real-time messages, manages the message buffering and provides access mechanism for the real-time data objects.

- Configuration Database
  This firmware module manages the flash memory subsystem. It manages the file system and the flash memory programming.

- Resource Manager
  A resource manager is a firmware module which keeps track of the resource usage in the system and control the resource allocation requests.

- Software Extension Modules Manager
  A software extension modules manager keeps track of the software extension modules installed in the system. It is responsible for enabling and disabling those modules.

- Device Drivers Manager
  The device drivers manager is the firmware module which knows the existence of all installed device drivers. It is the module responsible for enabling and disabling a device driver.

- Local Peripheral Bus Manager
  A local peripheral bus manager consists of two bus drivers which provides the basic input/output operation via those buses.

- Supervisor
  A supervisor in the main firmware module must initialize all other modules during start up of the system.

- Scheduler
  A scheduler is the most important part of the firmware because it controls the execution of tasks in the system.

The fundamental firmware components are described in detail in Section 3.3.2 to Section 3.3.12.

Figure 3.6 shows an overview of the firmware architecture. There are two sets of modules depicted in the figure which are not categorized as the fundamental modules; those modules are the Device Drivers and the Software Extension Modules. These modules are actually called the Firmware Extension Modules which will be described in more detail in Section 3.4.

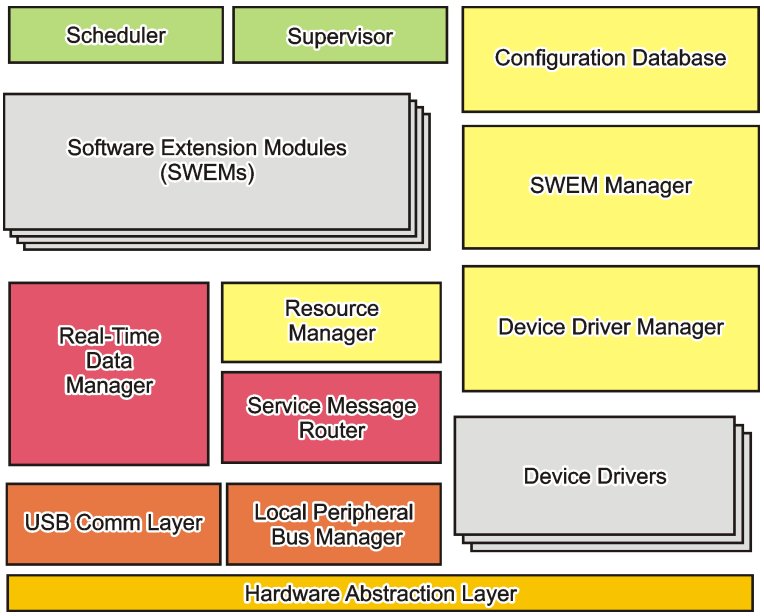

Figure 3.6: Firmware Architecture Overview

3.3.2 Hardware Abstraction Layer

The Hardware Abstraction Layer is a collection of system defines, macros and functions which are specific to the NXP LPC2888 microcontroller. This module allows the other firmware components on the higher level to perform hardware related operations. The hardware abstraction layer forms the lowest layer of the firmware which is used to communicate with the hardware. Below are the operations that are supported by hardware abstraction layer:

- Accessing General Purpose Input/output pins.
- Accessing integrated peripherals.
- Configuring the clock generation unit.
- Initializing the hardware timer.
- Configuring the microcontroller interrupt vectors.

This module is actually a compilation of various resources supplied by NXP as the developer of the microcontroller and some configuration related to the implementation of the RT-Motion USB board. This module is formed to centralized the hardware related code in order to create a portable firmware architecture. Should it be required that the firmware is ported to a new microcontroller, this module is the one which will need a great modification effort.

3.3.3 USB Communication Layer

The USB Communication Layer is the part of the firmware which controls the USB device controller in the microcontroller. It initializes the USB device controller, sets up the endpoints configuration and handles the USB enumeration process. After the enumeration process is finished then the task of USB Communication Layer is focused on the handling of USB messages from and to the host PC.

The USB Communication Layer is implemented in a discrete module, not integrated with the message handler for service channel and real-time channel. The decision was made due to the fact that the USB Communication Layer has a close dependency with the implementation of the USB Device Controller. Moreover, the message handlings on the service channel differs from those on the real-time channel. Figure 3.7 shows the handling of USB messages and the task delegation to Service Message Router and Real-Time Data Manager for each corresponding channel.

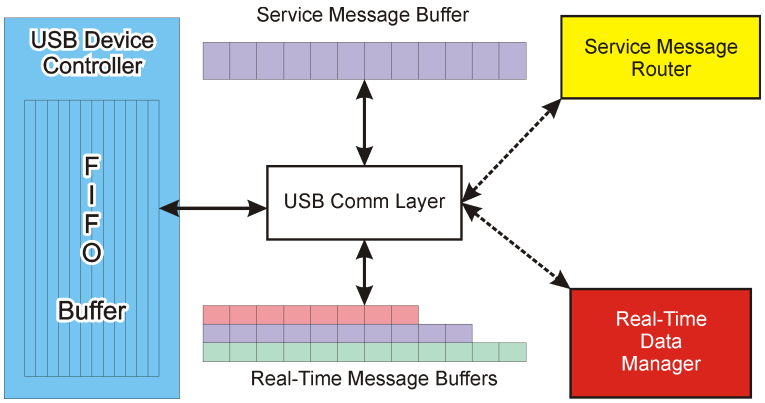

Figure 3.7: Incoming USB Data

Service Channel The USB Communication Layer maintains the pointer to the service message buffer which is maintained by the Service Message Router. Upon reception of data on the service channel, the USB Communication Layer will store the incoming message to the SMR's message buffer and notify the scheduler to execute the routing task of SMR.

The role of the USB Communication layer is to copy out the data from the internal FIFO buffer of the USB device controller to the message buffer owned by the SMR. The data copying is done in the interrupt service routine (ISR) context which must be short

as possible. At the end of the copying process, the USB Communication Layer does not contact the SMR directly to route the message. It only contact the Scheduler to schedule the execution of the routing task of the SMR. By utilizing this mechanism, the actual routing process and the service message interpretation and processing is not executed in the interrupt service context.

The handling of a message on the service channel is shown in Figure 3.8

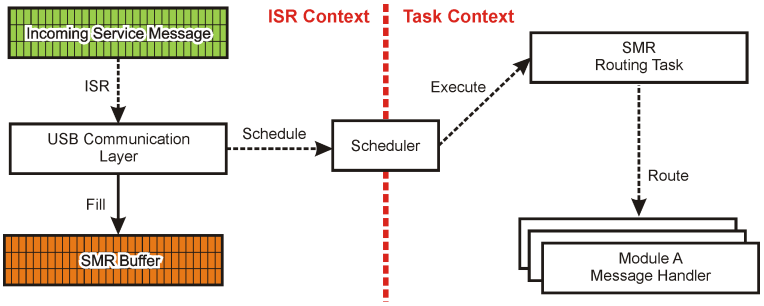

Figure 3.8: Service Message Handling by USB Communication Layer

Real-Time Channel Upon reception of a message on the real-time channel, the USB Communication Layer will pass the message to the *Real-Time Data Manager*. The role of the USB Communication Layer in handling the reception of a real-time message is limited to copying out the USB message from the internal FIFO buffer of the USB device controller and calling the Real-Time Data Manager to perform buffer swapping.

The USB Communication Layer maintain one buffer for each real-time message block size. The buffer is used to store incoming message and then swapped with the Real-Time Data Manager. The buffering of real-time message and the detail of the Real-Time Data Manager is described in Section 3.3.5.

3.3.4 Service Message Router

The Service Message Router (SMR) is a firmware module which routes service messages to various firmware modules and provide functions for sending a service message reply to the Host PC. The SMR can be regarded as the gateway of communication between the software running on the Host PC with various modules running on the RT-Motion USB board. Every module which need to receive a service message from the Host PC must first register itself to the SMR and submits a pointer to its service message handler function.

A message handler function is a code which is implemented by each module which expects to receive a service message from the host PC. This message handler function should recognize the incoming message and perform the associated task for each message and eventually the message handler function is obliged to send a reply to the host PC via the Service Message Router.

The SMR performs the message routing process in a system task which is triggered sporadically by USB Communication Layer upon successful reception of a service mes-

sage. The routing task of the SMR has the highest priority because in a way, the message routing still needs to have an execution warranty. So after a task is finished executing, the service message routing task will immediately executes. This design allows a complete isolation between the message handling process from the interrupt service routine of the USB controller. A message handler routine should not be executed from the interrupt service routine because there might be a case where a user develops a message handler which perform an unexpected behaviour and affect the whole system.

The message routing process is shown in Figure 3.9.

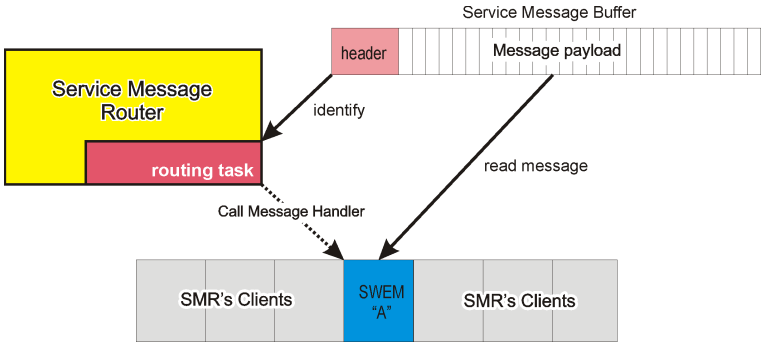

Figure 3.9: Service Message Routing Process

Service Message Buffering The required memory space for buffering the incoming and outgoing service messages is 512 bytes, shared for both service message. The memory space reserved for buffering the message is set to 512 bytes to match the maximum data payload size for bulk transfer mode [3, pp. 53]. The SMR only supports one service message to be routed at a time. Therefore if a new incoming message arrives while a message has not yet been replied by the message receiver, the new message will be ignored.

In the design stage of the SMR, a support for a simultaneous multiple service message processing was also considered. The multiple service message processing is not chosen because there is no real need for the application and it also introduces overhead both in memory usage and processing time.

Comparison to the original RT-Motion USB Firmware The original RT-Motion USB firmware also has a communication channel which is similar to the service channel, but it is called the configuration channel. That configuration channel purely functions to send configuration commands to the RT-Motion USB board. Some example of configuration commands are commands to enable or disable a motor amplifier channel, enabling or disabling onboard analog converters and any other configuration regarding input and output channels.

The interpretation and processing of the messages in the configuration channel is implemented directly inside the interrupt service routine. The approach of performing message processing directly from the interrupt service routine was also considered in the

design of the SMR. But the approach was abandoned because of the fact that service messages can serve a lot more complex commands which might take some time to be finished. Another reason is the fact that service messages does not need to be executed in a hard real-time.

3.3.5 Real-Time Data Manager

The Real-Time Data Manager (RTDM) is a firmware module which manages the real-time data objects (RTDO) and real-time messages. The user of the RT-Motion USB board must specify the mapping of the RTDO inside a real-time message during the communication configuration phase. The RTDM is designed to process an incoming and an outgoing real-time message differently. An incoming real-time message needs more buffering to ensure the validity of the message. While the buffering scheme of an outgoing real-time message is simpler.

The main difference between the buffering or an outgoing and incoming message lies on the updating process of the message buffer. The update process of an incoming real-time message is always done to the whole message buffer, which may take a long time, depending on the size of the message. The update process of an outgoing real-time message only updates one real-time data object at any time. This process is shorter and easier to manage than the incoming real-time message, thus it has a simpler message buffering scheme.

Incoming Real-Time Message The RTDM receives a bunch of RTDOs which are contained together in a real-time message and will save the value of those RTDOs in its internal buffer. Since the RTDM does not distribute the new value of the RTDOs automatically, the consumers of the RTDOs must actively request it. The design of the RTDM allows a very flexible message mapping mechanism, some RTDOs can be grouped together to form a real-time message. The RTDM itself supports more than one type of real-time messages which allows different message transfer schedules for different data, depending on the requirement of the application.

The RTDM allocates double memory buffer for each incoming real-time message type. The memory buffers are called *userBuffer* and *receiveBuffer*. As suggested by the name, each memory buffer serves its own function. The userBuffer is used to store the latest real-time message which is ready to be used for supplying the user with the latest RTDO value. While the receiveBuffer is used to store the incoming real-time message from the USB Communication Layer.

The management of the buffer inside the RTDM is designed to be done without any memory copying process. This is made possible by swapping the pointers to the buffers instead of swapping the content of the buffers. The passing of new real-time message from the USB Communication Layer also implemented by swapping the pointer to the buffers. The message buffer which is maintained by the USB Communication Layer contains the newly received message, while the receiveBuff maintained by the RTDM is not used by anyone, so the buffer swapping process can be done safely.

The swapping of message buffers between the USB Communication Layer and the RTDM is not a critical section in this design. It can be done directly by the USB Com-

munication Layer by calling the real-time message handler of RTDM from the interrupt service routine. The process itself is designed to be very simple, only swapping two buffer pointers and adjusting the flags of the buffers.

In the other side of the system, another swapping of message buffers pointers must be done between the userBuffer and the receiveBuffer. Whenever the receiveBuffer contains a fresh real-time message, the RTDM must swap the userBuffer with the receiveBuffer so that the user of RTDM can access the fresh RTDO values contained by the fresh real-time message. This buffer swapping procedure is done inside the RTDM function which supplies the user with RTDO values. Because the function is called by the user, which is inside the task of a software extension module, the process can be interrupted by a USB interrupt service routine. Thus this process is a critical section in the system, it needs to be protected by a USB interrupt disabling and enabling mechanism. However this critical section is only performing a buffer swapping procedure, which is very short. And this critical section only occurs every time a fresh real-time message is stored in the receiveBuffer. This design allows the USB interrupt to be disabled only for a very short time and at maximum only once every time a new real-time message is received.

But this design requires three buffers which are being swapped with each other to have the same size. Two buffers are allocated by the RTDM for each message type and one more buffer is allocated by the USB Communication Layer for copying out the real-time message from the USB device controller's internal buffer. In order to improve the memory usage efficiency of these message buffers, some standard message buffer sizes are defined. By having a standard message buffer size, the message buffer inside the USB Communication Layer can be used by more than one type of real-time message. As an example when there are two incoming real-time message types defined in the RTDM with size of 30 and 32 bytes, all memory buffers are allocated with the same size of 32 bytes, thus the total memory usage for the buffers are 160 bytes. Only one memory buffer with the size of 32 bytes is allocated by the USB Communication Layer.

Outgoing Real-Time Message The RTDM supports multiple types of outgoing real-time message. Each outgoing real-time message type can have different RTDO payloads and different size. This support of multiple outgoing real-time message type is designed to accommodate the need of performing transmission of RTDOs at different timing intervals. RTDOs which need to be transmitted together at the same time can be grouped into the same real-time message, while the others can form another real-time message.

The nature of USB communication requires that an outgoing message is already waiting in the USB controller's FIFO buffer when a read token is received from the Host PC. Therefore the RTDM must ensure that the next requested real-time message is submitted to the USB Communication Layer on time. This mechanism implies the requirement that RTDO is always informed in advance about the type of outgoing real-time message which the host PC expected to receive.

For each type of outgoing real-time message the RTDM maintains one message buffer. This message buffer is used to store the current value of the RTDOs contained by the real-time message. The process of updating the real-time message from the current value of RTDOs is done by the owner of each RTDO. Each outgoing RTDO must be assigned to one owner who is eligible of updating the value of the RTDO. This limitation is

introduced to prevent a conflict in the RTDO submission.

The process of updating the RTDO value inside the real-time message buffer is called *RTDO submission*. The RTDO submission must be done actively by the owner of the RTDO. This RTDO value submission process can be categorized as a critical section, because at any time the buffer of the outgoing message can be read for filling the USB device controller's outgoing buffer. The critical section is protected by disabling and enabling the USB interrupt during the memory copy operation from the owner's memory address to the RTDO's area inside the outgoing real-time message buffer.

Real-Time Message Task Triggering The RTDM is equipped with a feature that allows the execution of a specific task upon the reception of an incoming real-time message. This feature is called *real-time message task triggering*. The user can define that the RTDM will contact the scheduler to execute a specific task ID when the RTDM detects the arrival of a specific incoming real-time message type.

This feature is incorporated into the design of the RTDM to allow a synchronization mechanism between the real-time motion control algorithm running on the host PC with a task running on the RT-Motion USB board. By setting up a real-time message task triggering, the RTDM can detect the reception of a specific incoming real-time message to trigger the execution of a task. As an example the task is a task of one particular SWEM which is specially designed to process the values transmitted inside that particular real-time message. Then the execution of the task is synchronized with the reception of the real-time message and indirectly also synchronized with the execution of the motion control algorithm running on the host PC.

3.3.6 Configuration Database

The flexibility and configurability feature which are considered as the main objective of the firmware design requires a module which is capable of storing the configuration parameters which have been applied to the firmware. It is necessary to have a central configuration storage that is accessible by each module.

Actually an active configuration manager design was considered during the design stage, the active configuration manager receives all configuration parameters from the host PC, stores the configuration in flash memory and to applies the configuration to each corresponding module. But this kind of design was considered unpractical and introduced a more complex inter-module dependency. Instead of creating a configuration manager a less active solution was chosen during the design stage. The less active solution is then implemented as the Configuration Database.

The Configuration Database is a firmware module which is used to enable all modules in the firmware to save a block of configuration in the flash memory. It is designed to be a passive data storage manager which provides basic flash memory reading and writing functionality. The Configuration Database also keeps track of the list of configuration blocks which are stored in the flash memory. In order to keep track the of content of the flash memory, a data structure called *flash table of contents* is designed. The flash table of contents stores the mapping of configuration blocks which have been written into the flash memory. The flash table of contents is stored in the RAM but also written to the

flash memory to make sure that the flash memory is always consistent.

Because of the Configuration Database is only providing configuration block storage, then the configuration parameters are received by each module directly in the form of individual service messages from the host PC. Then each module applies those configuration parameters to the particular module. When the module need to store the configuration, the module can use the service which is provided by Configuration Database.

The reading of the configuration block is also done by the owner of the configuration block. The owner of configuration block calls a read configuration function with specifying its module identifier, then the Configuration Database returns the address of that configuration block inside the flash memory. The owner of the configuration block can read the configuration block as if from the random access memory.

A configuration block is basically a byte array with a maximum size of 2048 bytes. The configuration database only allow one configuration block per module, when a module saves a new configuration block, the existing configuration block of that module will be ignored. The total flash memory space available for configuration blocks is 48 kilobytes.

Flash Memory Subsystem The type of flash memory cell which is used in the NXP LPC2888 microcontroller is a NOR flash memory, which by principal allows bit by bit modification of the flash memory. The working principle of NOR flash memory allows every bit to be changed from 1 to 0 by programming it, but in order to revert the value of the bit to 1, the whole sector must be erased. By taking advantage of this important feature of NOR flash memory, we can design a flash memory storage which supports incremental programming. Incremental programming strategy is important to decrease the amount of flash memory erase operation because flash memory erase operation is both time and memory consuming consuming. We can only erase flash memory sector by sector.

The flash memory is organized into 64 KB large sectors and 8 KB small sectors. Having a total capacity of 1 MB, there are 15 large sectors and 8 small sectors. The organization of these sectors and corresponding address ranges is shown in 3.10. The usage of the flash memory sectors are arranged as follow:

- Small sector number 0: for flash table of contents
- Small sector number 1-6: for storing configuration blocks
- 15 large sectors: for saving the firmware image and future SWEM and Device Driver uploading.

Although the flash memory of NXP LPC2888 is based on NOR technology, the implementation of flash memory controller does not allow bit by bit modification to the flash memory area. This limitation is caused by the error correction mechanism which is implemented by the controller. Any bit which still have value 1 can be changed to 0 like the way NOR flash memory works, but this might produce an inconsistency problem to the error correction information. When the error correction information is inconsistent, then the flash memory controller will perform an incorrect bit recovery during the read operation.

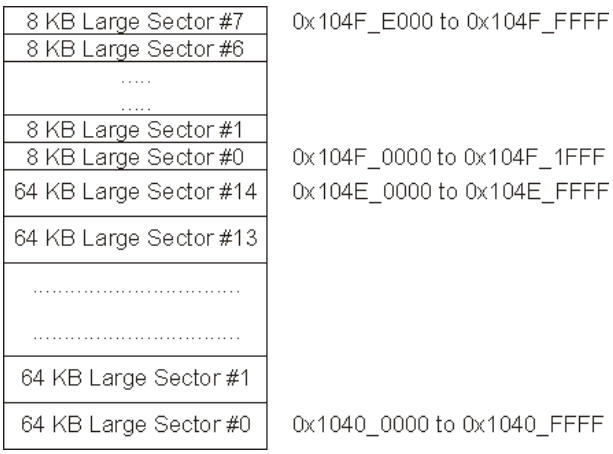

Figure 3.10: Flash memory sector organization

Unfortunately this error correction mechanism is not documented in the user manual of NXP LPC2888 and has caused a problem during the implementation of the Configuration Database. To avoid this error correction problem we have to design the flash memory system to always use 16 bytes alignment. This limitation must be applied to the design of the flash table of contents and the allocation of a new configuration block inside the flash memory. Each entry in the flash table of contents must be 16 bytes long to allow the incremental programming for the flash table of contents sector. The starting position of each configuration block also must be aligned to 16 bytes boundary.

3.3.7 Resource Manager

The Resource Manager is the firmware module which manages the access control of every system resources and provides the detailed information about every resource's access mechanism. Every device driver which provides additional resources must register the resources they provide to the Resource Manager. The Resource Manager stores the knowledge of all resources which are available in the system in a table which is called *Resource List*. In managing a resource allocation, resource manager updates the resource availability information inside the Resource List and also stores the resource allocation details in a table which is called *Resource Allocation Table*. This table stores the detail of each resource which has been allocated in the system, this table is mostly used for diagnosing a resource allocation problem.

Every resource which is listed in the Resource List have a unique resource identifier, along with the resource provider's identifier, and the resource sharing permission. The resource identifier is formed from the combination of the resource provider's location in the system and the unique resource identifier. The Resource Manager is designed to be able to handle resource sharing, where a resource can be used by more than one user. The resource sharing permission is defined by the provider of the resource during the resource registration process.

Every software extension module which needs to access a resource must request the

allocation of that particular resource to the Resource Manager. Resource Manager then will check the availability of that resource and the resource request will be granted if available. The user of a resource can also specify the whether it wants to use the resource exclusively or leave a resource sharing possibility for other user. After obtaining permission to access a certain resource, the new resource user should ask the Resource Manager for the detailed information about the driver of each resource which is needed to access the resource. After a resource user is granted access to a resource, the actual resource access is unsupervised by Resource Manager, each resource user writes and reads to the resource directly by accessing functions provided by the driver of the resource provider. This unsupervised access approach is used because we don't want to introduce an unnecessary overhead to the system. Unsupervised access means that after a resource user gain access to a resource, it has to use the resource accordingly. On the other hand, a resource user should not access a resource without allocating the resource or if the resource allocation is denied.

In general the role of the Resource Manager is to control the usage of resource and to be the mediator between the resource user and the resource provider. As a mediator the Resource Manager works by helping the resource user to find the needed resource and the detail information about the resource access mechanism. The Resource Manager helps the resource provider to publish the resource to the potential user.

3.3.8 Software Extension Modules Manager

The Software Extension Modules Manager is the firmware module which knows the existence of every *Software Extension Modules (SWEM)* which are embedded in the system. This module also keeps track of the entry point to each software extension modules and responsible to activate software extension modules when requested. The role of software extension module manager is only limited to activate the software extension module. The further configuration and controlling of each software extension module can be done directly to the software extension module by sending service messages.

The Software Extension Modules Manager maintain the list of active SWEMs in a data structure called *Active SWEM Table*. This Active SWEM Table can be saved to the Configuration Database to enable the initialization from flash procedure for the stand-alone mode. By getting the Active SWEM Table from the Configuration Database, the Software Extension Modules Manager can directly activate the listed SWEMs and triggers the SWEMs initialization.

The internal configurations of each SWEM is not stored by the Software Extension Modules Manager, but they are stored by each module which represents the good modular design of this architecture.

In addition to the SWEM management role, Software Extension Modules Manager also provides a mechanism for sharing data between the active software extension modules. One SWEM can allocate a SWEM Shared Data Object and any other SWEM which needs the information can read the shared data object value.

Software Extension Modules Manager provides various utility functions which are designed to be used by the SWEMs developer. These utility functions are mainly used to ease the development of administration part of the SWEMs.

Embedded Software Extension Modules Catalog The knowledge of the embedded software extension modules is composed in a data structure which is named Embedded Software Extension Module Catalog. The catalog must be updated every time a new software extension module is embedded to the system. Every entry of the *Embedded Software Extension Modules Catalog* contains the identifier of the SWEM and the function pointer to the SWEM's command handler entry point.

The reason of designing a Software Extension Modules Manager is to make the SWEM integration to the system easier. For integrating a new SWEM the developer only need to add a new entry into the Embedded Software Extension Modules Catalog. By registering the new SWEM to this catalog, the Software Extension Modules Manager can find that new SWEM and activate that SWEM on request.

3.3.9 Device Drivers Manager

The Device Driver Manager is a fundamental firmware module which manages and keeps track of every device driver which is embedded in the system. Device Driver Manager is responsible to find a device driver which has been embedded in the firmware and activate that device driver upon request from the Host PC.

The Device Driver Manager maintains the list of active device drivers and the function pointers related to each active device driver in a table called *Active Driver Table*. Every active device driver is accompanied with a pair of function pointers, which are the function pointer to the command handler of that particular device driver and a function pointer to the driver interface provider. When the user asks Device Driver Manager to save configuration to flash, Active Driver Table is saved to Configuration Database. By restoring this Active Driver Table from the flash configuration block, Device Driver Manager will be able to reinitialize every active device driver.

The individual configuration of each active driver is not maintained by the Device Driver Manager, but being managed and stored by each device driver. The role of Device Drivers Manager is limited to enabling and disabling the device drivers, further configuration commands are addressed directly to each device driver.

Device Driver Manager provides various utility functions which are designed to be used by the Device Drivers developer. These utility functions are mainly used to ease the development of administration part of the Device Drivers.

Embedded Device Driver Catalog The list of all device drivers which has been embedded is maintained in a data structure which is called *Embedded Device Driver Catalog*. The list contains the identifier of a device driver and the function pointer to the device driver's command handler entry point. Just like the Embedded Software Extension Module Catalog, this catalog also needs to be updated by the developer every time a new device driver is embedded to the system.

3.3.10 Local Peripheral Bus Manager

The Local Peripheral Bus Manager is the firmware module which is used to manage the buses which is used to connect the hardware extension modules to the RT-Motion USB

board. In the current implementation the RT-Motion Hardware Extension Modules can be connected to the main RT-Motion USB board via SPI bus and a parallel bus. The Local Peripheral Bus Manager is responsible to enable and disable the bus drivers as required by the user.

The SPI bus is not supported natively by the NXP LPC2888 microcontroller, so a hardware extension module called RT-Motion USB SPI Bridge is developed for providing SPI Bus support to the system. The SPI Bridge module is controlled by a driver which is called SPI Bus Driver. The developer of device drivers for interfacing hardware extension modules can perform SPI communication by calling the set of functions which is provided by the SPI Bus Driver.

For supporting the parallel bus, a Parallel Bus Driver is implemented. The SPI Bus Driver and the Parallel Bus Driver is designed to have a similar interface. By implementing a similar interface, the device driver creator can support device driver for both peripheral bus with small porting effort.

3.3.11 Supervisor

The Supervisor is the head of all other fundamental firmware module because the Supervisor is the one who must know the whereabouts of the other firmware modules. Supervisor is the firmware module which is responsible for the initialization of the firmware, and every other firmware modules.

Supervisor maintains the information about the board identifier and the operating mode of the particular board. Supervisor manages all system state transition of the RT-Motion USB firmware. Supervisor also provides a global error logging facility to let other firmware modules to submit their error messages. The error logging facility is an important feature for analyzing any initialization failure or for troubleshooting a runtime error in the system.

3.3.12 Scheduler

During the early stage of the research of this thesis some scheduler from off the shelf RTOSes are surveyed but those alternatives are considered to be overkill for this project and cost too much unnecessary overhead for the system. It is decided to build our own scheduler. The off the shelf scheduler experiment is covered in Section 2.5.1.

The scheduler is one of the most important fundamental firmware modules because it actually controls the execution of various system tasks. The working of Service Message Router also depends on the scheduler. Furthermore, any additional software extension module will also depend on the scheduler to execute its task correctly.

The scheduler in this system is designed as a simple non preemptive priority based scheduler. A non preemptive scheduler means that when a task is running, no other task can interrupt that task until it eventually pass the CPU time back to the scheduler. By choosing this scheduler model, the runtime overhead of the scheduler is lower than a more advanced preemptive scheduler.

The scheduler supports periodic tasks and sporadic tasks, a periodic task can be programmed to be executed periodically by the scheduler. On the other hand, a sporadic task is a task which execution is determined by the trigger sent by the user of that task.

Scheduler Operation Most scheduler are designed to utilize one hardware timer which is set to fire an interrupt specified interval. The interval between timer interrupt is called as the timer tick interval. Then the scheduler is designed to heavily rely on the timer interrupt service routine to keep track of the elapsed time and checking the task list for determining the next executable task [22]. Every time unit in the scheduler's implementation is defined in the timer tick interval.

Our scheduler is designed to use a different approach. The scheduler also utilize one hardware timer, but instead of setting up the timer to give interrupt every specified tick interval, the timer is setup to run continuously. The timer is setup as a free running counter which periodically overflows. An interrupt service routine is utilized for handling timer overflow events, but the interrupt service routine is only used to maintain the integrity of the task list after an overflow event occurred.

The time unit in our scheduler uses the counting frequency of the hardware timer. In this case the timer counting frequency of RT-Motion USB is set to 60 MHz, matching the core clock frequency of the processor. Thus the tick interval of the hardware timer is 16.67 ns.

Task Dispatcher Function The scheduler is equipped with a function which is responsible to check for any executable task at the moment, and execute the task when the execution schedule has arrived. This function is called the *task dispatcher function*. The task dispatcher function is called from the main loop of the main C program. Unlike most scheduler which performs the task list management process inside the timer tick's interrupt service routine, our scheduler perform the task list management process after finished executing a task. The task management process is called the task dispatcher function.

This scheduler design is proved to be efficient and able to provide high resolution schedule timing. The task dispatcher function of the scheduler continuosly compare the schedule of the next executable task with the value of the hardware timer. To optimize the performance of the task dispatcher function, the information of the next executable task and the execution schedule is always precomputed by the scheduler after finished executing one task.

Timing Diagnostics Feature The scheduler is also designed to have a feature for performing timing diagnostics. This feature can be activated during the runtime of the scheduler by sending an activation command via a service message. When the scheduler enters the timing diagnostics mode, it will use a different task dispatcher function which is equipped with various logging facilities. The timing diagnostics log is stored in the RAM of the RT-Motion USB board, and it can be fetched by the host PC by means of exchanging service messages. The scheduling events which can be logged are:

- Task Execution Delay
  Task execution delay is the time difference between the scheduled task release time with the actual task release time.

- Task Execution Time
  Task execution time is the duration of the task execution.

- Task Management Duration
  Task management duration is the time needed for performing the task management after finished executing a task.

The timing diagnostics log can be used to diagnose the schedulability of the task set given to the scheduler. This enables the user to fine-tune the schedule of the task set.

3.3.13 Firmware Modules Dependency Diagram

The diagram in Figure 3.11 shows the dependency relationship of the fundamental firmware modules. The inter-modules dependency is briefly described in the following section.

- SWEM depends on SWEM Manager, Scheduler, RTDM, Resource Manager, and Service Message Router.
  - Depends on the SWEM Manager for calling the utility functions.
  - Depends on the Scheduler for executing the periodic task.
  - Depends on the Resource Manager for allocating resources and getting the detail resource access mechanism.
- Device Driver depends on Device Driver Manager, Local Peripheral Bus Manager, Hardware Abstraction Layer, Resource Manager, Scheduler, Service Message Router, and Configuration Database.
  - Depends on the Device Driver Manager for calling the utility functions.
  - Depends on the Local Peripheral Bus Manager and the Hardware Abstraction Layer for accessing hardware.
  - Depends on the Resource Manager for allocating and registering resources.
  - Depends on the Scheduler for executing the optional background periodic task.
- SWEM Manager depends on Configuration Database and Service Message Router.
- Device Driver Manager depends on Configuration Database and Service Message Router.
- Resource Manager depends on Device Driver Manager, Configuration Database and Service Message Router.
  - Depends on the Device Driver Manager for getting the driver's interface provider.
- Local Peripheral Bus Manager depends on Resource Manager, Hardware Abstraction Layer, Configuration Database and Service Message Router.
  - Depends on the Resource Manager for allocating resources.
  - Depends on the Hardware Abstraction Layer for accessing hardware.

- Configuration Database depends on the Hardware Abstraction Layer and the Service Message Router.
  - Depends on the Hardware Abstraction Layer for programming flash memory.
- Service Message Router depends on USB Communication Layer and Scheduler.
  - Depends on the USB Communication Layer for handling USB communication.
  - Depends on the Scheduler for executing the routing task.
- Real-Time Data Manager depends on USB Communication Layer, Scheduler, Configuration Database and Service Message Router.
  - Depends on the USB Communication Layer for handling USB communication.
  - Depends on the Scheduler for handling the real-time message task triggering.
- USB Communication Layer depends on the Hardware Abstraction Layer for accessing hardware.

Generally speaking, the Supervisor depends on every module because the Supervisor starts all system modules after power up, thus it must know about the whereabouts of the modules and how to initialize those modules. Most modules depend on the Service Message Router for performing communication with the host PC via the service channel. Most modules depend on the Configuration Database for storing their configuration block.

3.3.14 State of the Firmware

The firmware module that takes care about the system state transitions is the Supervisor module. The Supervisor will initialize the board and manage state of the firmware throughout the execution of the platform. Figure 3.12 shows the system state diagram of the modular RT-Motion USB firmware.

The description of each system state is described in the following section.

Microcontroller Unit (MCU) Initialization This state is entered by the firmware right after the board is powered on. The initialization process of the hardware includes the generation of the microcontroller's clock tree and initialization of all internal resources. The first fundamental module which is initialized in this state is the Supervisor, which in turn will initialize every other firmware fundamental modules.

Idle After all initialization has been successfully done, the firmware will enter this idle state. In this state the firmware will check the operating mode setup of the board. Depending on the value, the firmware has to either wait for service message from host PC or advance to the next state.

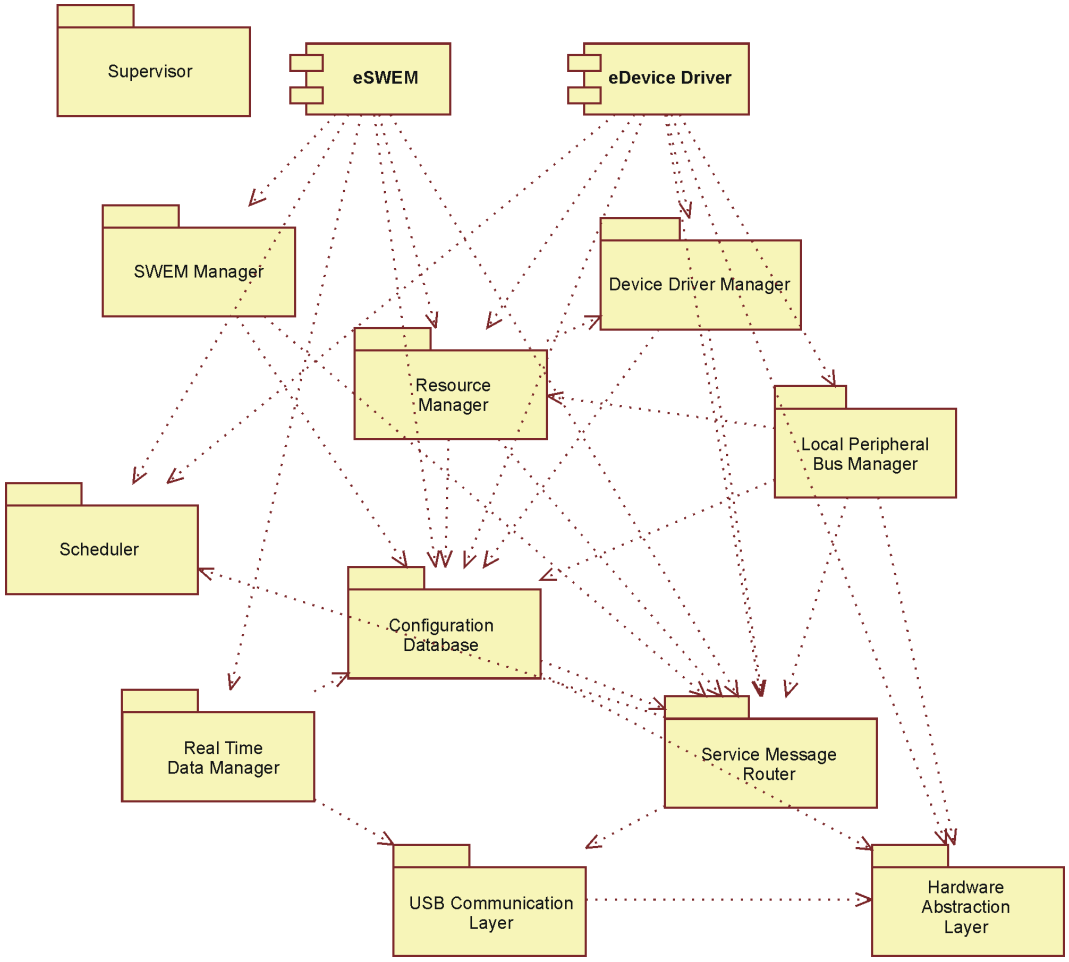

Figure 3.11: Firmware Modules Dependency Diagram

Hardware Configuration State The hardware configuration state is the first phase of the configuration sequence. In this phase the user of the board has to specify the device drivers which need to be activated in the particular board. Each activated device driver should also be configured and initialized during this state. The device driver's configuration can also be fetched from the configuration block stored in the Flash memory. To advance to the next configuration state all of the enabled device drivers must be correctly initialized at the end of this state.

Communication Configuration State This configuration state is used to set the required real-time data objects and place it into real-time messages. At the end of this state, the Real-time Data Manager will allocate the memory space required for buffering the real-time messages.

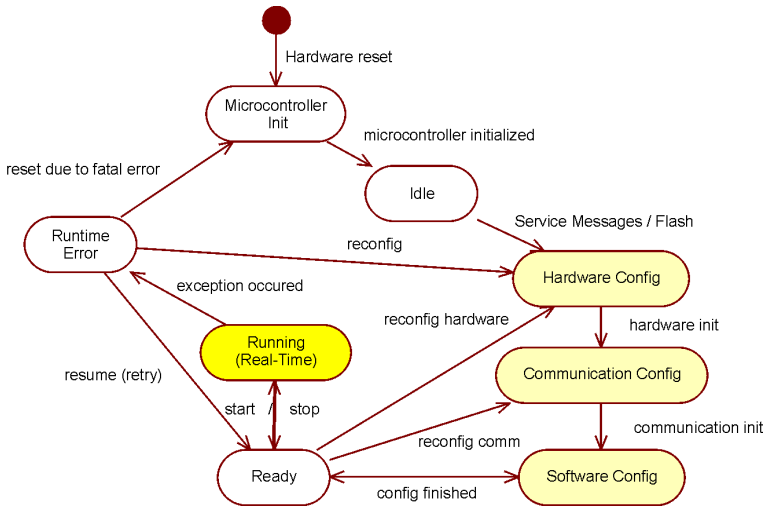

Figure 3.12: System State Diagram

Software Configuration State The software configuration state is the last phase of the configuration sequence. In this phase the user can choose which software extension modules are to be activated. The individual configuration of each software extension modules are also configured in this phase. The configuration of software extension modules includes the assignment of real-time data objects and required resources allocation, which in principle requires the outcome of the two preceding configuration phases.

Ready State The ready state is the state where the firmware is sitting idle waiting for a service message to start the application. When such message arrived, the Supervisor will contact the Scheduler to enable the Scheduler and start executing all periodic tasks.

Running State The scheduler is running and executing all tasks which exist in the Task List. In this state the system must meet the real-time constraints.

Runtime Error State When the firmware encounters a critical error which forces a halt of operation it will go to the Runtime Error State. In this state the Host PC can query the RT-Motion Board about the detail of the error and decide the recovery action.

3.3.15 Firmware Initialization Modes

The Modular RT-Motion USB Firmware can be initialized from these conditions: the board has no configuration at all, the board is configured as slave mode and the complete system configuration is found in the Configuration Database and the board is configured as stand alone mode.

Empty Configuration If the board has no pre-saved configuration block, the initialization process of the board will wait for a sequence of services messages sent by the

Host PC.

Slave Mode with pre-saved Configuration If the board has a pre-saved configuration which is configured as Slave mode, the board will wait for the instruction to apply the pre-saved configuration. If Supervisor receives a service message instructing the board to use the pre-saved configuration, the board will fetch the configuration from Configuration Database and start initializing the board as configured in the pre-saved configuration.

Stand Alone Mode If the board has a pre-saved configuration which is configured as Stand-alone mode, the board will immediately fetch the configuration from Configuration Database and initialize the board accordingly.

3.3.16 Firmware Execution Contexts

The firmware only has two active interrupt service routines, which are:

- USB communication
  The USB communication is handled in an Interrupt Service Routine (ISR), which basically copies out the data from the USB Controller's FIFO buffer to the corresponding manager's buffer.

- Scheduler's Timer Overflow
  The timer which is used for the scheduler's timer is set as a free running counter which eventually will overflow, to manage the scheduling list; an ISR is implemented for handling the timer's overflow event.

The main part of the scheduler is executed from the idle loop of the firmware. Since the scheduler is a non preemptive scheduler, only one active task can be executed at a time. An active task cannot be preempted by any other task. An active task can only be preempted by interrupt service routines.

The system by default will have at least one active task which is registered by the Service Message Router, this task is used to perform the service message routing process and the priority of that task is set to a high priority. Other additional tasks can be added to the system by the active software extension modules. A device driver can also optionally add a task for implementing a background task for supporting the device driver's inner working.

3.3.17 Firmware Memory Map

The majority of the firmware modules can be executed directly from flash, but the performance of executing code from flash is around 20 percent slower than executing the code from RAM. So in order to gain the maximum performance some modules will have to be executed from RAM, and the others can be executed from flash. Generally all pieces of code which are only used to initialize and configure the firmware can be executed directly from the flash memory.

The Scheduler, the Real-Time Data Manager, the Service Message Router, and the USB Communication Layer are all important modules which is greatly used during the real-time running mode of the firmware, so these modules need the maximum performance that can be delivered by the microcontroller. These modules are configured to be executed from RAM. Table 3.1 shows the memory footprint of the firmware modules. The total amount of RAM which has been used is around 39 KB, which means we still have some room free RAM for executing crucial modules from RAM.

All other modules can be executed from flash memory to save the limited RAM space. Software extension modules and device drivers are by default designed to be executed from Flash because the amount of available RAM might not be enough to accommodate all device drivers and software extension modules. However when a SWEM requires a very high performance, it can be specified to be executed from RAM.

Table 3.1: Modular RT-Motion USB Memory Footprint

Module Name	Code@Flash	Code@RAM	Data@RAM
Configuration Database	4,696		3,096
Device Driver Manager	2,948		3,851
Hardware Abstraction Layer	3,936		3,328
Heap Area			8,192
Local Peripheral Bus Manager	904		4
Main C Program	80		
Parallel Port Bus Driver	2,620		12
Resource Manager	2,432		3,588
Real-Time Data Manager		6,452	668
Scheduler		3,808	4,768
Service Message Router		1,636	776
SPI Bus Driver	2,204		3
Supervisor	3,412		504
SWEM Manager	3,044		514
USB Communication Layer		968	40
USB HAL	5,682		132
Total	31,958	12,864	26,010

3.3.18 Implementation Details

The design of Modular RT-Motion USB Firmware which is described in this section is implemented in C language. The development process uses the Keil Microcontroller Development Kit. The optimization used in this project is *Optimization Level 3* and *Optimized for Time*. The source code of the firmware is organized such that the implementation of each module is grouped into a separate folder.

Table 3.2 shows the total lines of code used for implementing the Modular RT-Motion USB Firmware, as calculated by *cloc* [4].

Table 3.2: Lines Of Code Count of Implementation

Language	Files	Blank	Comment	Code
C	42	4,664	6,851	12,994
C/C++ Header	74	1,370	2,631	2,909
Assembly	2	121	194	125
SUM:	118	6,155	9,676	16,028

3.4 Firmware Extension Modules

Firmware extension modules are modules which can be integrated to the firmware to provide additional functions to the firmware. The objective is to let as many developers as possible create firmware extension modules to the system. In order to be able to write a firmware extension module, the developer must adhere to the extension modules format specification. There are two kinds of firmware extension modules which are supported in the Modular RT-Motion Firmware. They are:

- Software Extension Module
  This format of extension module is used to develop a custom algorithm to be executed on the firmware. A software extension module can have real-time data exchange with the Host PC.
- Device Driver
  A Device driver is a format of extension module which is specially designed to control an external hardware module, to be the interface of a hardware module.

In the current implementation of the firmware, the firmware extension modules have to be linked together with the basic firmware and uploaded as a single image to the RT-Motion USB board. However, support for uploading the firmware extension modules on-the-fly can be added in the future. The on-the-fly uploading here does not mean that the software extension module can be uploaded during real-time operation mode, but during the configuration mode of the firmware. This on-the-fly software extension module uploading will enable the addition of software functionalities without recompiling the firmware.

3.4.1 Software Extension Module

A software extension module (SWEM) is an extension module that can deliver some custom additional functionality to the system. A particular SWEM can be designed to obtain information, process the information and then deliver that information to the outside world. The framework provides software extension modules with mechanisms to get input information from either real-time USB channel or external hardware modules (with the help of device drivers) and to deliver their output to the outside world.

A software extension module can be made to be executed periodically or sporadically. The sporadic execution can be made upon reception of a real-time trigger message from the host PC or from other SWEM. The task code which is going to be executed sporadi-

cally or periodically must always take into account the real-time constraints which must be met by the system.

One software extension module can have multiple instances of data and configuration which share the same code. Each SWEM instance can be executed on different interval and with different configurations. The data of each software extension module instance is encapsulated in a SWEM Instance table; each kind of SWEM can have its own instance data type. Before a SWEM can be executed the user must configure all required configuration of each instance, the configuration might include the assignment of resource ID, the assignment of real-time data objects and the real-time message task triggering. The required resources are allocated during the initialization of the SWEM instances. A successful resource allocation might lead to the process of obtaining the device driver interface to control each particular resource.

Every software extension module should implement the service message handler to be able to communicate with the Host PC via service channel. The service channel communication is used to exchange configuration commands. Every software extension module is able to save its configuration block to the flash memory with the help of the Configuration Database.

One SWEM can share information with another SWEM by allocating a shared data object from the Software Extension Modules Manager. Every shared data object must be assigned during the configuration phase and checked during the initialization, so that those shared data objects can be used directly during the runtime of the application.

The general structure of a software extension module is shown in Figure 3.13.

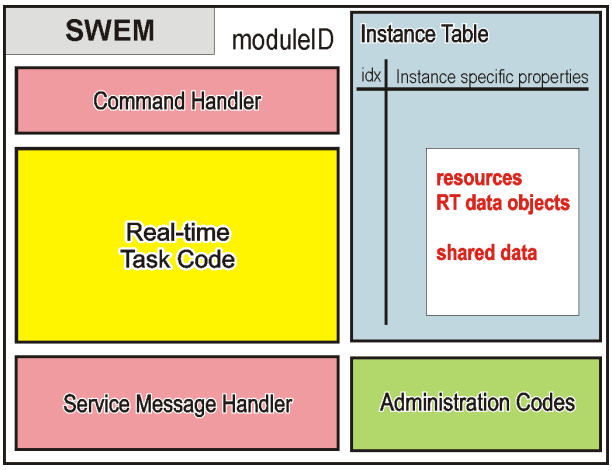

Figure 3.13: Structure of a Software Extension Module

3.4.2 Device Driver

A device driver is an extension module that is used to give an abstraction layer to enable software developers to access hardware functionality easily. The functions provided by a device driver are specific to the device it is representing. A device driver can contact

the Resource Manager for requesting to allocate resources that it needs to operate and eventually registering resources that it is providing.

A device driver is designed to control one specific type of hardware with specific extension module attachment method, whether it is via SPI bus, via GPIO port or internal peripherals. But the system supports more than one extension modules with the same type to be attached to RT-Motion USB board, which introduces the device driver instance term. One device driver interface represents one physical hardware extension module with a unique address. In addition to the unique address per device driver instance, the developer of a device driver can also store any instance specific configuration or data in the device driver's instance table.

Every device driver must at least implement one type of device driver interface, which is actually the way a device driver can provide its services to the user of the device driver. The user of a device driver knows which kind of interface type the device driver should be supporting, thus the user knows what functions is provided and can be called to perform any hardware specific operation that the driver is controlling. A device driver can also implement more than one device driver interface, which will give freedom for the user of the device driver to choose which interface it is going to access.

Figure 3.14 shows a setup of RT-Motion USB connected with 3 physical hardware extension modules of the same type (RTM_BLDC Motor controller) and also the internal motor amplifier. There are three device drivers which need to be enabled to control all those modules:

- RTM_BLDC_Par Driver
  This driver is used to control the RTM_BLDC which is connected to the 16-bit GPIO port.

- RTM_BLDC_SPI
  This driver is used to control the two RTM_BLDC modules which are connected to the SPI Bridge. Because the driver must control two physical modules, these two modules are uniquely identified by the instance number inside the device driver. Each driver instance stores the module specific configuration.

- Internal_MotorController Driver
  This driver is used to control the internal motor controller which is integrated in the RT-Motion USB itself.

All these three drivers can implement the same device driver interface which eventually enables the flexibility for the user to choose which kind of hardware is going to be used without thinking about the different programming interface.

The developer can also design a device driver which emulates the functionality of a device; therefore, from the application point of view, the device emulation driver still provides the same interface as the driver for the physical device.

The general structure of a device driver is shown in Figure 3.15.

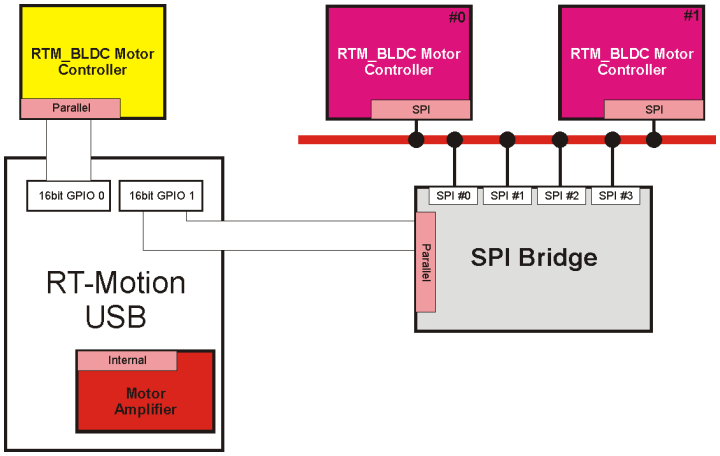

Figure 3.14: Device Drivers Instances

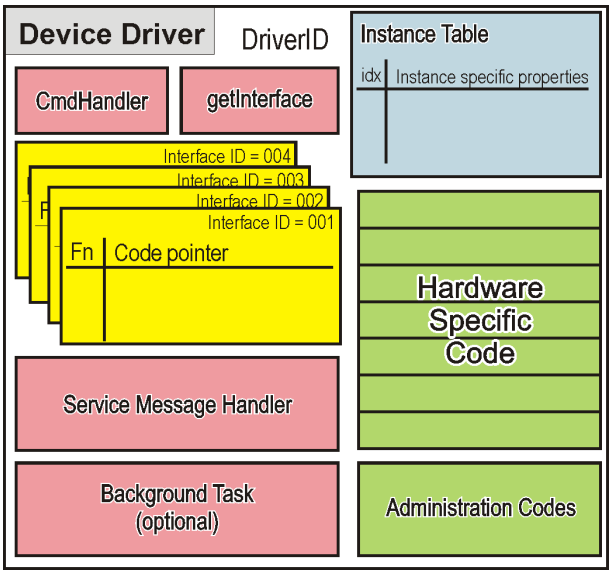

Figure 3.15: Structure of a Device Driver

3.5 Host PC Supporting Framework

3.5.1 USB Driver

The USB Device Driver for the RT-Motion USB board is developed by Sait Izmit during the completion of the graduation project [11]. The USB Device Driver was developed to run on Vanilla Linux Kernel, Linux Kernel with RTAI extension and Linux Kernel with Xenomai extension. A driver wrapper has also been implemented to enable the device driver to be used in Windows operating system environment.

The USB Device Driver which is used in the Modular RT-Motion USB Software

Framework is based on the original device driver for the RT-Motion USB board. The original USB driver which has been developed for the RT-Motion USB board has a maximum packet size of 64 bytes, meanwhile according to the USB specification the maximum size of messages on bulk transfer is 512 bytes, so some modifications have been made to enlarge the maximum packet size of the USB endpoints to be 512 bytes. Setting the maximum packet size to 512 bytes enables us to use larger message which can lead to more efficient communication.

3.5.2 User API

The User Application Programming Interface (User API) for the new Modular RT-Motion Software Framework must be developed to enable the developer of Host PC Motion Control Application to communicate with the new firmware.

The User API will send appropriate service messages to enable the User Application in the Host PC to communicate with the Firmware.

3.5.3 Configuration Editor

The Configuration Editor is an application which is designed to be executed in the Host PC, which is used mainly to help the user of the RT-Motion USB board in configuring the board. This application will guide the user in the board configuration sequence.

In addition to providing guidance for configuring the board, the configuration editor can also create a configuration file which mirrors the completed configuration of that particular board. This configuration file can then be used to configure the board directly from the User Application.

3.5.4 Real-Time User Application

The real-time user application is the user application which is running in the Host PC to communicate with the RT-Motion USB Board and typically used to control the board, get sensor information from the board, perform motion control algorithm and supply the latest actuation value to the RT-Motion USB board.

This user application can perform USB communication with the RT-Motion USB board using both the service channel and the real-time channel. Every action is done by calling functions which are provided by the User API.

4 Experimental Results

This chapter covers the experiments which have been performed as case studies in this research project. In general the performance of the developed framework is analyzed thoroughly and the system overhead is measured.

4.1 Scheduler's Task Switching Time

4.1.1 Testing Methodology

A software extension module (SWEM) has been developed to measure the performance of the scheduler. The developed SWEM has a periodic task which is used to simulate execution time by calling a delay function. This particular SWEM is called *SWEM SchedTester*. SWEM SchedTester can be used to simulate the execution time of task with resolution of 1 microsecond resolution. It is possible to make up to 16 instances of SWEM SchedTester with different intervals, release times, execution times and priorities. These features make SWEM SchedTester suitable to be used as a measurement tool. The scheduler of the Modular RT-Motion USB Firmware will be referred using the term *ModRTM Scheduler*.

In performing task switching time measurement, SWEM SchedTester is enabled and configured to run with different number of instances, in which each instance creates one additional task to the system. In this experiment, the routing task of the Service Message Router is deactivated in order to purely measure the scheduler's performance. So the total number of tasks managed by the scheduler is the same as the total number of SWEM SchedTester instances being activated. Each SWEM SchedTester instance is configured to have an execution time of 100 microseconds and all are released at the same time and have the same priority.

The timing measurement is performed by toggling a GPIO pin to indicate the active process at that moment, and then the GPIO pin is probed by an oscilloscope. The oscilloscope used for the experiment is an Agilent MSO6034A which is equipped with 4 analog input channels. Figure 4.1 shows a local zoom of the oscilloscope screen capture which displays the details of the task switching process with 2 active tasks in the system. The first two channels represent the execution time of task 1 and 2 respectively; the third channel is unused in this experiment; the fourth channel represents the activity of the task dispatcher function of the scheduler. When a task is active the signal seen by the oscilloscope is high, and low when inactive. When the signal on channel 4 is high, that means the task dispatcher function of the scheduler is active. The task dispatcher function will become inactive after it finished performing the task management process following a task execution, or there is no executable task at the moment.

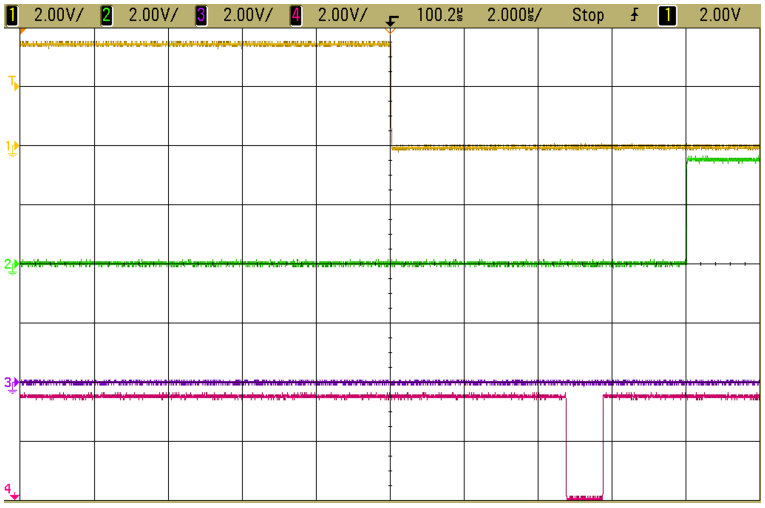

Figure 4.1: Oscilloscope Screen Capture of Task Switching Between 2 Tasks

4.1.2 Results of Modular RT-Motion USB Scheduler

In general the task switching time is the sum of task management time, task dispatching time and the time between two consecutive task dispatcher executions. The task management is linearly influence by the number of tasks because the task management process needs to scan through the whole task list array to find the next executable task. However, the task dispatching time and the time between the task dispatcher executions is independent of the number of task. The time between two consecutive task dispatcher executions is a fixed value of around 1 microsecond, which can be accounted for the microcontroller's overhead for branching into a function, toggling a GPIO pin, returning from a function and doing an infinite loop.

The result of the task switching measurement experiment can be observed in detail in Table 4.1. The unit of time measurement is microseconds. The first column in the table shows the number of tasks which are active in the system. The second column represents the duration of the task dispatching process before start executing a task. The third column represents the duration of the task management process after executing a task. The fourth column represents the amount of time spent in the main loop of the C program between two consecutive task dispatcher executions. And finally the fifth column shows the total switching time of the scheduler for the given number of active tasks.

4.1.3 Comparison with other platforms

In this section we investigate the task switching time of the schedulers which used by the other platforms. The first platform we investigate is the legacy RT-Motion USB Firmware. And then the result from the Modular RT-Motion USB Firmware and the result of the scheduler used by the Keil RTX Real-Time Kernel are also investigated.

Table 4.1: ModRTM Scheduler Task Switching Time

Nr of Tasks	Dispatcher	Management	Main Loop	Task Switching Time
2	2.3	4.8	1	8.1
3	2.3	5.4	1	8.7
4	2.3	6.2	1	9.5
5	2.3	7.1	1	10.4
6	2.3	8.0	1	11.3
7	2.3	8.8	1	12.1
8	2.3	9.7	1	13.0
9	2.3	10.6	1	13.9
10	2.3	11.5	1	14.8

4.1.3.1 Legacy RT-Motion USB Firmware

The legacy RT-Motion USB firmware does not have native support for multi-tasking because it is not equipped with any scheduler. In order to execute a periodic task, the developer must utilize the internal timer peripherals provided by the microcontroller and place the code inside the interrupt service routine.

For testing purposes a timer interrupt is set up to virtually execute 4 tasks sequentially. This kind of setup is actually being used in the Robot Eyes project to execute 4 channels of PID controller every 500 microseconds (2 KHz frequency). According to past experiences, RT-Motion USB board cannot execute the 4 channels PID controller on a higher frequency than 2 KHz. For that reason, a delay function is used to easily simulate the CPU time used by the tasks.

Figure 4.2 shows the virtual task switching time between PID controller channels. The measured switching time is obviously very small, around 0.5 microsecond. This extremely low switching time is possible because the PID controllers are actually executed sequentially one after another within a *for* loop. But of course this static implementation of periodic task execution does not offer the flexibilities which are achievable using the ModRTM Scheduler. That every task must be executed with the same interval and sequentially.

4.1.3.2 Modular RT-Motion USB Firmware

In order to imitate the execution of 4 channels PID controller running at 2 KHz frequency, the Modular RT-Motion USB Firmware is configured with one active SWEM SchedTester. The SWEM SchedTester is configured to have four instances, each having an execution time of 100 microseconds and all released at the same time. Figure 4.3 shows the oscilloscope screen capture of the execution of the first three tasks and the activity of the scheduler's task dispatcher function on the fourth input channel of the oscilloscope.

The experiment showed in Figure 4.3 shows that the scheduler is able to execute 4 tasks each with execution time of 100 microseconds at frequency of 2 KHz while having around 60 microseconds idle time in every cycle. The execution of 4 tasks with the

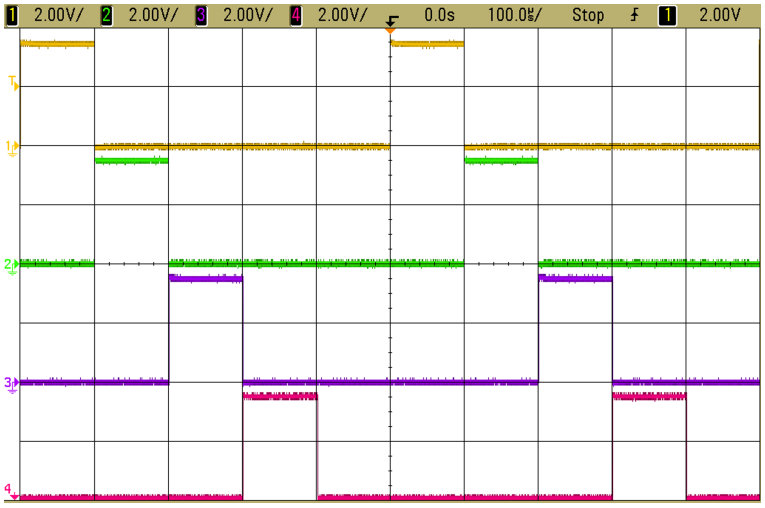

Figure 4.2: Legacy RT-Motion USB Periodic Task Execution

same release time means that only the first task is executed on time, while the three following tasks are late. But overall those tasks are still schedulable because before the cycle restarts, all tasks have been finished executing.

The experiment also showed that the task switching time for this kind of configuration is around 10 microseconds, which confirms the result of task switching time scaling experiment showed in Table 4.1.

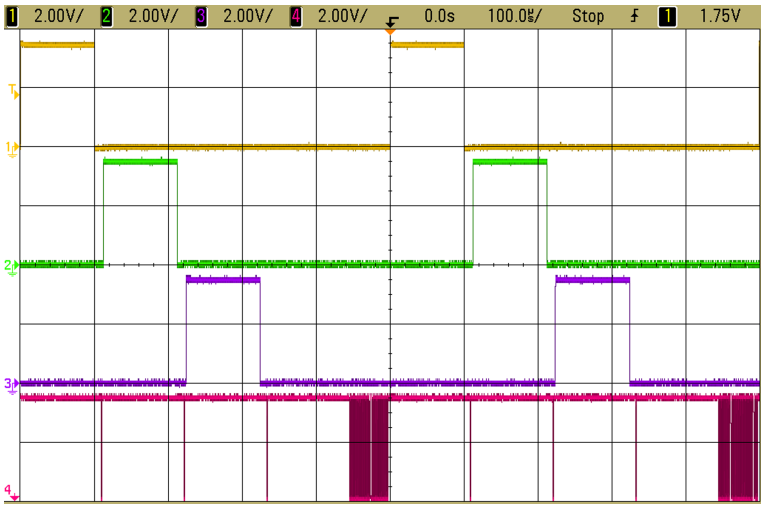

Figure 4.3: ModRTM Scheduler 4 Tasks @ 2KHz Same Release Time

4.1.3.3 Keil RTX

As a measure the compare the performance of our own-build scheduler, a small number of scheduler performance measurements have been executed on Keil RTX Real-Time Kernel platform. The Keil RTX is configured to run in the non preemptive mode, with a timer tick interval of 50 microseconds. The timer tick interval is used as the base value for defining all time units in the system, including the task intervals and user timers.

Keil RTX is designed to perform one interrupt service routine every timer tick interval [14]. This interrupt service routine is used to keep track of the elapsed time, administer its task list, and to determine which task to be executed next. This timer interrupt has an average execution time of around 20 microseconds. In general, if the timer tick interval is set to 50 microseconds, the scheduler introduces 40 percent CPU overhead. A larger timer tick interval must be selected in order to reduce the scheduler's overhead. This means that Keil RTX will not be able to schedule tasks in the same time resolution as Modular RT-Motion USB Scheduler.

Figure 4.4 shows a screen capture of the oscilloscope while doing the task switching experiment using Keil RTX with 2 active tasks. The task switching time is fluctuating. When the task switch occurs at the same time as the timer tick interrupt service routine execution, the task switching time takes around 20 microseconds longer. The detailed measurement result of task switching time under Keil RTX scheduler can be observed in Table 4.2. The unit of time measurement is microseconds. The results also show the scaling of Keil RTX performance when loaded with various amount of active tasks. There are two kinds of task switching time shown in Table 4.2, the Normal task switching time is the measured task switching time when the task switch occured without being interrupted. The Interrupted task switching time shows the task switching time where in the middle of the switching process a timer tick interrupt occurred.

Compared to the task switching time of ModRTM Scheduler, the Keil RTX have a better scaling to the number of tasks. But the overhead of the timer tick interrupt service routine is not found in the task switching time of ModRTM Scheduler, resulting to a much more predictable task switching time.

Table 4.2: Keil RTX Scheduler Task Switching Time

Nr of Tasks	Normal	Interrupted	ISR Overhead
2	11.5	31.3	19.8
3	12.0	31.7	19.7
8	13.9	33.5	19.6

4.2 Priority Driven Non Preemptive Scheduler

In this section, an experiment is designed to show the effectiveness of the priority driven non preemptive scheduler. This scheduler is used as the basic of the ModRTM Scheduler.

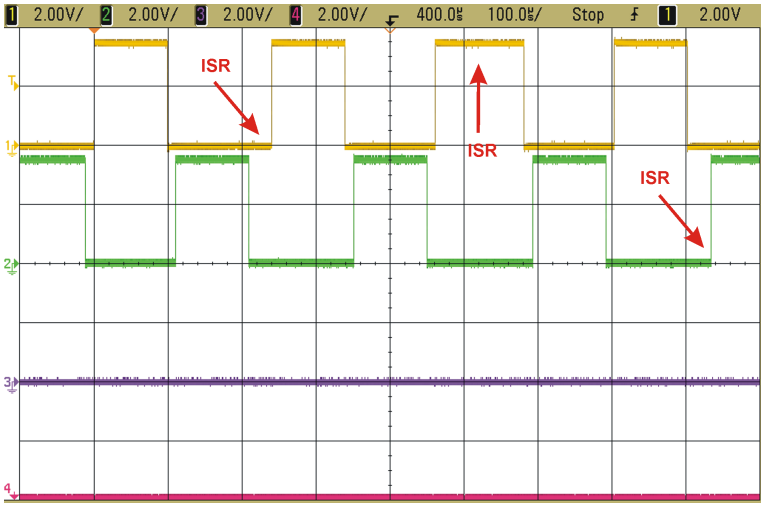

Figure 4.4: Keil RTX Task Switching Time

4.2.1 Testing Methodology

The experiment utilizes again the SWEM SchedTester. For this particular purpose, three instances of SWEM SchedTester are created. In order to simulate the priority scheduler, different execution times, different initial release time schedules, different intervals and different priorities are assigned for each SWEM SchedTester instance. The configuration of SWEM SchedTester creates a task set as shown in Table 4.3 and illustrated in Figure 4.6.

Table 4.3: Task Set of SWEM SchedTester With Different Priorities

Task ID	Execution Time	Interval	Priority
1	2 ms	5 ms	1 (lowest)
2	1 ms	10 ms	2 (higher)
3	3 ms	15 ms	3 (highest)

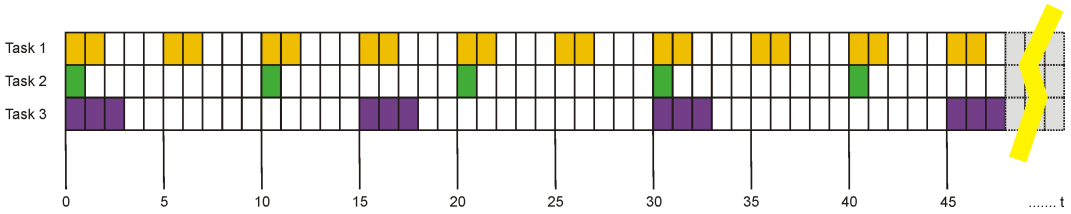

Figure 4.5: Illustration of Task Set With Different Priorities

4.2.2 Result

The actual execution of the three tasks that have been scheduled is presented in Figure 4.6. Task 3 which has the highest priority is always executed first. This could lead to the delay of other task with the same schedule, which will only be executed after task 3 has been completed. The task execution delay is described in more detail in Section 4.4. The priority driven scheduler provides a better scheduling possibilities and eventually leads to a more flexible schedule. By default the system has a task with a very high priority for handling the routing process of service messages. By giving a high priority to the service message routing task, we can make sure that the service message will be routed right after a task finished executing.

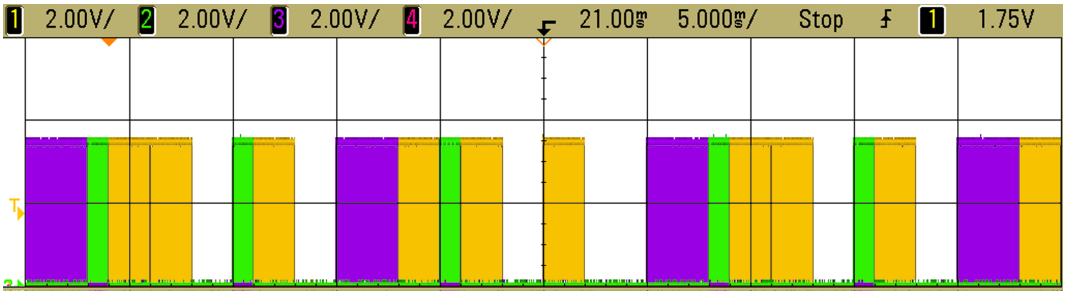

Figure 4.6: Actual Schedule of the 3 tasks in the Priority Scheduler Experiment

4.3 Service Message Routing Task Execution

This section focuses on the investigation of the reception of a service message, the firing of the SMR's routing task, the switching time and the actual execution of the service message task. This experiment also shows a drawback of non preemptive schedulers, where a higher priority task, the SMR's routing task, cannot interrupt the execution of another task with a lower priority.

The screen capture of the oscilloscope showing the service message reception event can be observed in Figure 4.7. The first input channel of the oscilloscope is connected to a GPIO pin which represents the execution of SWEM SchedTester task. The second input channel is connected to a GPIO pin for the service message routing task. The third input channel is connected to a GPIO pin used to detect the interrupt service routine of the USB controller, when the controller receives the USB frame containing the incoming service message. The last input channel is connected to a GPIO pin which shows the activity of the scheduler.

The switching time from task 1 to the SMR's routing task is shorter than ordinary task switching. This is possible because the process of *firing* a sporadic task allows the scheduler to bypass the task management routine at the end of the execution of task 1. The task management routine is bypassed because after the execution of task 1, the scheduler already knows what is the next executable task, which is set by the firing of

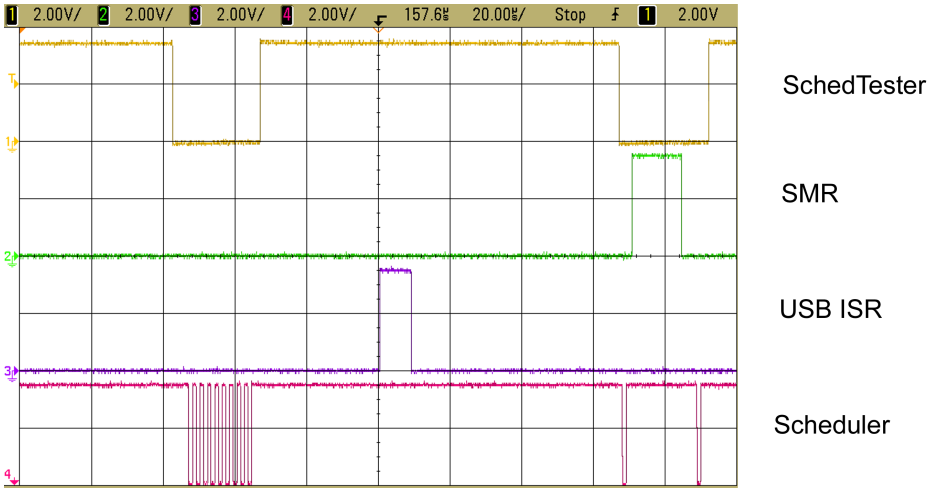

Figure 4.7: Reception of a Service Message

the sporadic task. The measured task switching time to a sporadic task is around 3.7 microseconds, independent of the number of active tasks in the system.

4.4 Task Execution Delay

This section investigates the task execution delay of the Modular RT-Motion USB Scheduler. The task execution delay is defined as the time difference between the scheduled release time of a task and the actual execution time. The variations of task execution delay can be influenced either by the overhead of the scheduler itself or the schedulability of the active tasks. This section focuses on the measurement of task execution delay which is caused by the overhead of the scheduler itself. For detecting task execution delays which is caused by the poor schedulability of the active task set, the scheduler is equipped with a timing diagnostics feature. This timing diagnostics feature is described in Section 3.3.12.

4.4.1 Testing Methodology

The measurement of the task execution delay of each executable task is performed by getting the current value of the timer used by the scheduler and comparing it with the scheduled task release time which is recorded in the task list. If the time difference observed is larger than certain customizable threshold, the value is stored into the scheduler's timing log. This log table is designed to store the first 500 entries and is then sent to the Host PC and saved to a file for further processing.

For testing the task execution delay, the same SWEM SchedTester is used. However, instead of releasing all tasks at the same time, each task is now given a different release time with enough time spacing between tasks to achieve a schedulable task list. All tasks have the same interval of 1 milliseconds, the same execution time of 100 microseconds,

and the same priority. The task set used for this experiment is shown in Table 4.4.

Table 4.4: Task Set of SWEM SchedTester With Different Release Time

Task ID	Initial Task Release Time (us)	Interval (us)	Execution Time (us)	Priority
1	0	1000	100	1
2	125	1000	100	1
3	250	1000	100	1
4	375	1000	100	1
5	500	1000	100	1
6	625	1000	100	1
7	750	1000	100	1
8	875	1000	100	1

4.4.2 Results

Table 4.5 presents the measurement result of the task execution delay. The unit of time measurement is microseconds. This experiment uses a task list which is schedulable, which means that the execution of a task followed by the scheduler's task management process is finished executing before any other task in the system becomes executable. In this kind of scheduling condition, the task execution delay is not influenced by the number of active tasks in the system.

The task management process is executed after a task is finished executing, which will keep the task execution delay at minimum if the task set given to the scheduler allocated proper time spacing among tasks. The required task management time allocation can be seen in Table 4.1. The variation between the maximum and minimum task execution lateness observed in Table 4.5 is mainly caused by the position in time when the Scheduler's Task Dispatcher function is called by the main C function.

Table 4.5: ModRTM Scheduler Task Execution Delay

Nr of Tasks	Average	Minimum	Maximum
1	1.78	0.87	2.70
2	1.78	0.83	2.73
3	1.77	0.83	2.70
4	1.78	0.83	2.73
5	1.79	0.83	2.73
6	1.78	0.83	2.73
7	1.79	0.83	2.70
8	1.78	0.83	2.73

4.5 USB Communication

Some measurements of USB subsystem performance were performed during the experiment stage of the project. The execution time of various functions of the Service Message Router and the Real-Time Data Manager were measured. The execution time of the USB device controller's interrupt service routine (ISR) for handling incoming USB message in service channel endpoint and real-time channel endpoint were also measured.

4.5.1 Timing of USB ISR for Service Channel Receive

An experiment for measuring the execution time of the USB ISR for a service channel endpoint has been done by taking the value of the scheduler's timer before and after the execution of the ISR. The measured value is then stored in the timing diagnostics log of Scheduler. The timing diagnostics log is then fetched by the host PC. Table 4.6 presents the timing measurement of the USB interrupt service routine for handling message reception on the service channel and the scaling of performance with regard to the size of the message being received. The timing measured here are in time units of microseconds. The task which is done inside the USB interrupt service routine is mainly a FIFO buffer copying from the USB controller to the Service Message Router's message buffer and a function call to the Scheduler to fire the execution of the Service Message Router's routing task.

Table 4.6: USB ISR for Service Channel Receive Execution Time

Message Size	Execution Time (μs)	Time/Byte
12	7.75	0.65
16	8.28	0.52
32	9.89	0.31
64	13.65	0.21
128	20.05	0.16
256	32.85	0.13
512	58.45	0.11

The results show that the execution time scales with the size of the message, but the processing time per byte is getting more efficient as the message size grows larger. The improvement of efficiency shows that the time needed for performing the buffer administration and task firing procedure does not depend on the size of the message. The buffer administration and task firing process can be regarded as the message processing overhead of SMR.

When the RT-Motion USB board is working in a stand alone mode and running a high frequency controlling task onboard, in most cases the USB communication will be disabled or mostly idle. It will only be used for performing some administrative issues or getting diagnostics logs. Thus, this measurement is primarily intended for measuring the RT-Motion USB board when working in slave mode, performing a real-time USB data exchange at a lower frequency of around 1 KHz. This means that the execution time of the USB interrupt service routine for service channel is acceptable because with

lower control frequency we have less strict timing constraints.

While running in the slave mode configuration and performing a real-time USB data exchange, the service channel endpoint is also expected to be mostly idle or at most sending short USB messages of around 12 to 16 bytes. This means that the required time by the USB ISR for handling service channel message reception will be as low as 8.28 microseconds.

4.5.2 Timing of USB ISR for Real-Time Channel Receive

The execution time of the USB ISR for a real-time channel endpoint has been measured by taking the value of the scheduler's timer before and after the execution of the ISR. The measured value is then stored in the timing diagnostics log of the Scheduler, which is then fetched by the host PC. Table 4.7 presents the timing measurement of the USB ISR for handling message reception on the real-time channel and the scaling of performance with regard to the size of the received message. The timing measured are in time units of microseconds. The USB interrupt service routine includes the buffer copying from the USB controller's FIFO buffer and a call to the Real-Time Data Manager's incoming message handler function, which performs the message buffer management.

Table 4.7: USB ISR for Real-Time Channel Receive Execution Time

Message Size	Execution Time (μs)	Time/Byte
16	8.72	0.54
32	9.73	0.30
64	13.53	0.21
128	19.93	0.16
256	32.77	0.13
512	58.33	0.11

The results show that similar to the USB ISR for processing incoming service messages, the execution time of incoming real-time message processing time in the USB ISR also scales with the size of the message. The message processing overhead of real-time message is also comparable to the service message, even though the series of operation quite differs. For processing a real-time message, the USB Communication Layer does not need to call the Scheduler to fire any task but directly call the message handler function of the RTDM.

The USB interrupt service routine can also call a function to fill the buffer of the USB device controller for the outgoing endpoint of a real-time channel in response to an outgoing real-time message request packet. This kind of request packet is only 7 bytes long, composed of 6 bytes of header and 1 byte of packet identifier of the requested packet. The experiment which is done is requesting a message with size of 30 bytes as an example. The total execution time of the USB ISR for processing this particular incoming message is around 12.38 microseconds. This execution time includes the time for copying out the message body from the USB controller's buffer, identifying the message, and filling the buffer of the outgoing real-time channel endpoint with a message of 30 bytes.

4.5.3 Timing of USB Device Controller's Buffer Filling

The filling process of the USB device controller's FIFO buffer with a message to be delivered to Host PC is handled by USB Communication Layer. This process is implemented in a function which can be called by either a Real-Time Data Manager or a Service Message Router.

The measurement is taken by utilizing the same method as the previous measurement, by capturing the scheduler's timer value and saving the time difference in the Scheduler's timing diagnostics log table. The detailed measurement which shows the scaling of performance with regard to the size of the message is shown in Table 4.8

Table 4.8: Execution Time of USB Buffer Filling

Message Size	Execution Time (μs)	Time/Byte
16	2.80	0.18
32	3.92	0.12
64	6.20	0.09
128	10.72	0.08
256	19.80	0.07
512	37.93	0.07

From this measurement result we can conclude that the amount of time needed for filling the USB buffer depends on the size of the message and some message preparation process overhead. This message preparation overhead makes the scaling of time required per byte to decrease as the size of message increases. In general message with size of 64 bytes may provide the best balance for most application, because the time required per byte is relatively low enough.

4.5.4 Real-Time Data Manager

4.5.4.1 Timing of Getting a Real-Time Data Object Value

The process of getting a real-time data object (RTDO) value is important to be measured because periodically the consumer of the RTDO could call this function from the periodic task of a SWEM. The periodic task of a SWEM must meet the real-time constraint of the system.

Table 4.9 shows the measurement results of get RTDO value functions which are provided by the RTDM to allow the consumer of an RTDO to get the latest data. RTDM provides two functions for getting the latest RTDO value. The two functions differ in such a way that one of the functions is optimized for operating with 32-bit sized data and the other function can serve the value for RTDO with any data size. The measurements show that the execution time for object size of 3 bytes is slower than for object size of 4 bytes, as the result of the optimized function for an object size of 32 bits. The optimization for an object size of 32 bits is implemented because 32-bit variables can be considered to be a common variable size.

Table 4.9: Execution Time of Getting an RTDO Value

Object Size	Execution Time (μs)	Time/Byte
3	2.97	0.99
4	1.62	0.40
8	3.61	0.45
16	3.72	0.23
32	4.10	0.13
64	5.07	0.08

4.5.4.2 Timing of Getting a Real-Time Data Object Value

Table 4.10 shows the measurement result of functions provided by the RTDM to allow the owner of an RTDO to submit its RTDO value to be delivered to the Host PC. A special function for optimizing 32 bits sized RTDO is also implemented. Again the execution time of submitting an RTDO with an object size of 3 bytes is slower than RTDO with an object size of 4 bytes because the submission of a 4 bytes sized RTDO object utilizes the optimized version of the function.

Table 4.10: Execution Time of Getting RTDO Value

Object Size	Execution Time (μs)	Time/Byte
3	3.52	1.17
4	1.93	0.48
8	3.67	0.46
16	3.77	0.24
32	4.22	0.13
64	5.15	0.08

These experiments show that when the SWEM needs to exchange more than one RTDO, grouping of all required data in a *struct* data type with a larger RTDO size can improve processing time efficiency.

4.5.5 Service Message Router

Timing of the Routing Task This experiment investigates how much time is used by the routing task to route a service message to a firmware module and let the module process that particular message and send a reply to the host PC. For this experiment, the module which is used as the destination of the service message is the Configuration Database. A message was sent in order to get the table of contents of the Flash Memory with several requested sizes. This would give an impression on the scaling of the system.

The measurement result is shown in Table 4.11 with a time units of microseconds. But this timing measurement result also includes the message processing and replying time used by the destination module.

These results show the total time used for processing a service message from the routing of the message until the message is replied by the message recipient. The required

Table 4.11: Execution Time SMR Routing to Configuration Database TOC Request

Reply Size	Execution Time (μs)	Time/Byte
16	10.50	0.65
32	15.40	0.48
64	18.47	0.28
128	15.07	0.19
256	38.25	0.15
512	64.67	0.13

time is again acceptable and within the limits because by nature the amount of service message transaction is expected to be very low during the real-time execution of the software. Most of the service message transactions are to be performed during the configuration phase of the firmware in which real-time constraints are not an issue.

4.6 Conclusion

Based on the above mentioned results we can say that the task switching time of Mod-RTM Scheduler is more predictable than Keil RTX's scheduler and even though it scales worse, but for small amount of tasks the performance is good. The experiment which investigates the effectiveness of the priority scheduling shows that the scheduler is proved to be able to schedule tasks with different priorities correctly.

Meanwhile the experiment with the service message routing task execution shows that the sporadic task execution works well and able to provide non preemptive interrupt-like behaviour for service messages. As a result of the higher priority given to the routing task of SMR.

The task execution delay measurement shows that the task execution delay is not influenced by the number of tasks. And finally the USB communication timing measurement shows that the size of message influences the efficiency of the transmission time per byte. An correctly chosen message size can deliver the best performance to meet each platform application.

Conclusions and Recommendations

5

In this last chapter we conclude this master's thesis. First, we make a summary of the development of this research and evaluate the goals of this research. Finally, we give a list of recommendations which can be used to improve the software framework and outline for future development of this research.

5.1 Conclusions

The main contribution of this project is the introduction of a great deal of extendability to the existing RT-Motion USB framework. Originally, the default firmware of the RT-Motion USB is a static firmware which only provides a real-time distributed I/O functionality, leaving the rest of the processing power unused. A framework which allows the user of the RT-Motion USB board to easily develop an onboard executable custom code and a driver code to interface an external hardware has been designed. These features will allow the RT-Motion USB platform to advance along with the broadening of the platform application.

Another important contribution is the highly configurable nature of the Modular RT-Motion USB Software Framework. This allows Philips Applied Technologies to deliver the RT-Motion USB board with a firmware which incorporates all software extension modules and device drivers which have been implemented. Furthermore, the user can configure the board individually to meet their own needs.

The conclusion of this thesis project is that the designed modular software framework is able to provide a highly extendable and configurable framework while maintaining the overhead of the framework at minimal. The memory footprint and performance overhead of the framework is larger than the original firmware. It is, however remains within the required limit. Furthermore, the gained flexibility gives a much more significant contribution.

Our project showed that the Modular RT-Motion USB Software Framework can expand the RT-Motion USB to be a general purpose motion control platform.

5.2 Recommendations

Even though all aims of the project have been successfully met, there will always be a room for improvement. Below is a list of several recommendations to improve the future development of the software framework.

Extension Modules Integration The way of integrating custom developed software extension modules and device drivers is still very limited and depending on Philips Ap-

plied Technologies. This limitation and dependency to Philips Applied Technologies is due to the fact that the software extension modules and device drivers must be compiled together with the firmware by Philips Applied Technologies. However, the design has accommodated the room for improvement. In the future further research can be performed to let the user compile the software extension modules and device drivers separately and upload the binary file of the extension modules on-the-fly.

Faster USB Communication The USB device controller inside the NXP LPC2888 also has an internal DMA controller to offload the CPU from filling and getting message from its FIFO buffer. The implementation developed in this thesis project is not using DMA, which leaves room for further performance improvements.

Faster Microcontroller The RT-Motion USB board relies mainly on the NXP LPC2888 because of the unique High Speed USB 2.0 controller which is integrated in the NXP LPC2888. Newer microcontroller models are mostly only equipped with a Full Speed USB 2.0 controller, which is much slower than a High Speed USB 2.0 controller. But recently NXP released a brand new microcontroller which is based on the ARM9 core running at 180 MHz and equipped with a High Speed USB 2.0 controller. The new microcontroller from NXP is called NXP LPC3131. This microcontroller can be considered as the base platform for a future RT-Motion USB when higher performance is required.

The software framework developed in this thesis project is organized in such a way that the porting to a new microcontroller will not require too much effort. In order to port this software framework to another microcontroller the developer only need to modify modules which are related directly to the hardware, such as the Hardware Abstraction Layer the USB Communication Layer, and the device drivers.

Requirements Specification

A

This section describes the requirements specification which was formulated at the beginning of the project.

A.1 Firmware Foundation

- The RT-Motion USB has two operating modes: Initialization Mode and Real-Time Execution Mode.
- USB Boards identification by software based addressing
- The board should be capable to handle up to 2 KHz control frequency on the Host PC side, in the current robot arm controlling application, the control loop is executed on the PC at control frequency of 1 KHz.

A.2 Configuration Manager

- The configuration manager can save the configuration to the on-chip flash for the future use on request.
- Perform self test to ensure that the configuration is valid and executable on the physical hardware before goes into running mode.

A.3 Software Extension Modules Manager

- Provide a standard specification for developers in writing software extension modules.
  - Input/output mechanism
  - USB Communication
  - Parameter Setting
  - Parameter Tuning
- The integration of software extension module is done during the firmware compilation process and uploaded to the board in the same binary image as the firmware.
- Each software extension modules can have a list of tunable variables/parameter/configuration that can be tuned by the Configuration Manager from the Host PC on the fly or set during the initialization mode.

- Software Extension Module can have some configuration which can be saved in the Flash memory
- Potential Firmware Extension Modules
  - Local control loops
  - Signal processing and conditioning tasks such as speed acquisition and filtering out measurement noise
  - Software based encoder counter
  - Driver for add-on local peripheral
- The board should be able to run local control loop at high frequency.

A.4 Scheduler

- Real-time scheduler to manage the real-time execution of the RT-Motion USB.
- Keeps track of the CPU time usage of a software extension module for diagnostic purpose

A.5 Resource Manager

- Enabling dynamic resource management to make a better use of the on-board resources.
- Control the allocation of resources to software modules to avoid resource conflicts.

A.6 USB Communication

- Configurable real-time data exchange to optimize the utilization of the communication channel.
- Dynamic mapping data between data exchange packet and their corresponding modules which can be defined during initialization mode.
- Configuration data channel to send commands between the Host PC and the USB boards
- Host PC should also be able to communicate with the USB boards on the fly (while running in a motion-control environment) to tune some settings

A.7 Local Peripheral Bus Manager

- Provide some standard driver model and standard communication protocol to support broad range of add-on hardware functional modules in cooperation with hardware developer.

A.8 Diagnostic

- Measurement of add-on software modules' execution time
- Measurement of USB communication delay, which can be measured as the time elapsed between the submission of packet to the buffer and the sending process.
- Place message index on USB packets.
- Tracing services
  - Tracing services can be implemented as a software extension module; the amount of tracing buffer is very limited since the amount of available RAM is also very limited.

A.9 USER API

- User API to ease the development of the control application on host PC.

Off the shelf RTOS

B

Below is the list of off the shelf real-time operating systems that have been surveyed.

Table B.1: RTOS comparison

	FreeRTOS	Keil RTX	uCOS/II
License	Free	included in MDK	>8500 €
Scheduler	Yes	Yes	Yes
- Preemptive	Yes	Yes	
- Cooperative	Yes	Yes	Yes
- Round Robin		Yes	
Message queues	Yes		
Message mailboxes	No	Yes	Yes
Semaphores	Yes	Yes	Yes
Mutex	Yes	Yes	Yes
Stack overflow protection	Yes	Yes	
Memory management	No	Yes	Yes (Fixed block)
Resource manager	No	No	
Task Management		Yes	Yes
- Real-time control		Yes	Yes
- Create			Yes
- Delete			Yes
- Change priority			Yes
- Suspend			Yes
- Resume			Yes
Timer services		Yes	Yes
Context switch time		<5 usec @ 60MHz	
ARM7TDMI port	Yes	Yes	Yes
LPC2xxx port	Yes	Yes	Yes
LPC2888 port	No	No	Yes

Bibliography

[1] ARM, *ARM processor instruction set architecture*, http://www.arm.com/products/CPUs/architecture.html.

[2] Jan Axelson, *USB complete third edition*, Lakeview Research, Madison, Wisconsin, 2005.

[3] Intel Lucent Microsoft NEC Philips Compaq, Hewlett-Packard, *Universal serial bus specification*, USB Implementers Forum, Beaverton, Oregon, 2000.

[4] Northrop Grumman Corporation, *Cloc - count lines of code*, http://cloc.sourceforge.net/.

[5] Ellisys, *Ellisys USB explorer 200 analyzer*, http://www.ellisys.com/products/usbex200.

[6] Steve Furber, *ARM system-on-chip architecture second edition*, Addison-Wesley, Harlow, England, 2000.

[7] EtherCAT Technology Group, *EtherCAT - ethernet for control automation technology*, http://www.ethercat.org/en/ethercat.html.

[8] Christopher Hallinan, *Embedded linux primer*, Prentice Hall, Upper Saddle River, New Jersey, 2007.

[9] CAN in Automation, *CiA draft standard 301*, (2002).

[10] CAN in Automation (CiA), *CANopen*, http://www.can-cia.org/index.php?id=171.

[11] Sait Izmit, *Real-time USB-based distributed control architecture for robotics applications*, MSc Thesis TU Delft, Delft, The Netherlands, 2007.

[12] Keil, *Keil RL-RTX timing specifications*, http://www.keil.com/support/man/docs/rlarm/rlarm_ar_timing_spec.htm.

[13] _____, *Keil RL-RTX user's guide*, http://www.keil.com/support/man/docs/rlarm/rlarm_ar _artxarm.htm.

[14] _____, *Keil RL-RTX user's guide - timer tick interrupt*, http://www.keil.com/support/man/docs/rlarm/rlarm_ar_timer_tick.htm.

[15] _____, *Realview® Compilation Tools*, http://www.keil.com/arm/realview.asp.

[16] _____, *The software floating-point library, fplib*, http://www.keil.com/support/man/docs/armlib/armlib_cihfebih.htm.

[17] Robert Schwebel Luotao Fu, *RT PREEMPT HOWTO*, http://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO.

[18] Micriμm, *μc/usb-bulk overview*, http://www.micrium.com/products/usb/usb-device/usb-bulk.html.

[19] MontaVista, *Montavista linux 6 - the new approach to embedded linux development*, http://www.mvista.com/product_detail_mvl6.php.

[20] Total Phase, *Beagle USB 480 protocol analyzer*, http://www.totalphase.com/products/beagle_usb480.

[21] _____, *Beagle USB protocol analyzer market comparison*, http://www.totalphase.com/products/compare/beagle_usb.

[22] Michael J Pont, *Patterns for time-triggered embedded systems*, Addison-Wesley, Harlow, England, 2008.

[23] Linux-USB Project, *USB testing on linux*, http://www.linux-usb.org/usbtest.

[24] Wind River, *Wind river vxworks*, http://www.windriver.com/products/vxworks/.

[25] NXP Semiconductor, *LPC2888 Errata Sheet*, (2008).

[26] RTAI Team, *RTAI - the realtime application interface for linux from DIAPM*, https://www.rtai.org/index.php.

[27] Xenomai, *Xenomai: Real-time framework for linux*, http://www.xenomai.org/index.php/Main_Page.