LAPPEENRANTA UNIVERSITY OF TECHNOLOGY
Faculty of Technology
LUT Energy
Electrical Engineering

Juha Salli

RTOS FRAMEWORK FOR REAL-TIME CONTROL SYSTEM

Examiners of Thesis: Professor D. Sc. (Tech) Tuomo Lindh
D. Sc. (Tech) Julius Luukko

Instructors of Thesis: D. Sc. (Tech) Julius Luukko

ABSTRACT

Lappeenranta University of Technology
Faculty of Technology
LUT Energy
Electrical engineering

Juha Salli

RTOS Framework for Real-Time Control System

Master's thesis

2009

88 pages, 15 figures, 12 tables and 11 appendices

Examiners: Professor D. Sc. (Tech) Tuomo Lindh
D. Sc. (Tech) Julius Luukko

Keywords: Real-time operating systems, embedded software development, hard real-time, periodic execution, hardware abstraction layer, information hiding.

This thesis is done as a complementary part for the active magnet bearing (AMB) control software development project in Lappeenranta University of Technology. The main focus of the thesis is to examine an idea of a real-time operating system (RTOS) framework that operates in a dedicated digital signal processor (DSP) environment. General use real-time operating systems do not necessarily provide sufficient platform for periodic control algorithm utilisation. In addition, application program interfaces found in real-time operating systems are commonly non-existent or provided as chip-support libraries, thus hindering platform independent software development. Hence, two divergent real-time operating systems and additional periodic extension software with the framework design are examined to find solutions for the research problems.

The research is discharged by; tracing the selected real-time operating system, formulating requirements for the system, and designing the real-time operating system framework (OSFW). The OSFW is formed by programming the framework and conjoining the outcome with the RTOS and the periodic extension. The system is tested and functionality of the software is evaluated in theoretical context of the Rate Monotonic Scheduling (RMS) theory. The performance of the OSFW and substance of the approach are discussed in contrast to the research theme. The findings of the thesis demonstrates that the forged real-time operating system framework is a viable groundwork solution for periodic control applications.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
Teknillinen tiedekunta
LUT Energia
Sähkötekniikka

Juha Salli

Reaaliaikakäyttöjärjestelmäkehyksen kehitys reaaliaikaiseen ohjausjärjestelmään.

Diplomityö

2009

88 sivua, 15 kuvaa, 12 taulukkoa ja 11 liitettä

Tarkastajat: Professori, TkT. Tuomo Lindh
TkT. Julius Luukko

Hakusanat: Reaaliaikakäyttöjärjestelmät, sulautetun ohjelmiston kehitys, kova reaaliaikaisuus, jaksolliset algoritmit, laiterajapinnan häivytys, informaation piilotus.

Tämä diplomityö on tehty Lappeenrannan teknillisessä yliopistossa, osana digitaali- ja säätötekniikan laboratorion aktiivimagneettilaakerisäätöohjelmiston kehitysprojektia. Työssä tutkitaan reaaliaikakäyttöjärjestelmään liitettävän ohjelmistokehyksen soveltuvuutta ja käytettävyyttä digitaalisessa signaaliprosessorissa. Yleiskäyttöiset reaaliaikakäyttöjärjestelmät eivät välttämättä mahdollista tarkasti jaksollista suoritusta vaativien ohjausalgoritmien käyttöä. Lisäksi ohjelmiston kehitysrajapinnat yleensä puuttuvat tai ne ovat sisällytetty laitteen ohjelmistotukikirjastoihin, mikä vaikeuttaa laiteriippumattomien ohjelmistojen kehitystä. Ratkaisun löytämiseksi tutkitaan kahta erillistä reaaliaikakäyttöjärjestelmää, jaksollisen suorituksen mahdollistavaa ohjelmistoa sekä suunnitellaan ohjelmistokehys vastauksena tutkimusogelmiin.

Tutkimus on suoritettu tarkastelemalla valittuja reaaliaikakäyttöjärjestelmiä, muodostamalla järjestelmän suorituskykyvaatimukset sekä suunnittelemalla reaaliaikakäyttöjärjestelmään pohjautuva ohjelmistokehys. Ohjelmistokehys on muodostettu alusta alkaen ohjelmoidusta sovellusrajapinnasta, joka on liitetty yhteen valitun reaaliaikakäyttöjärjestelmän sekä erillisen jaksollisen suorituksen mahdollistavan ohjelmiston kanssa. Ohjelmoitu reaaliaikakäyttöjärjestelmäkehys on koestettu todellisessa laiteympäristössä. Koestuksesta syntyneitä tuloksia on tarkasteltu teoreettisessa viitekehyksessä sekä suhteessa tutkimusteeman kysymyksiin. Tutkimustulokset osoittavat että työn tuloksena aikaansaatu reaaliaikakäyttöjärjestelmäohjelmistokehys sopii pohjaksi jaksollista suoritusta vaativille nopeille ohjaus- ja säätösovelluksille.

Preface

This master's thesis is done as a part of research project of Laboratory of Control Engineering and Digital Systems in Lappeenranta University of Technology. I am privileged and grateful for having obtained a scholarship granted by the Support Foundation of the Lappeenranta University of Technology as support for completion of the thesis.

I would take the opportunity to express my gratitude towards D.Sc (Tech) Riku Pöllänen for starting the research project and kindly offering an opportunity to finish up my master's thesis. Additionally, I would like to express my acknowledgement towards professor D.Sc (Tech) Tuomo Lindh for being the inspector of the thesis. Especially I would like thank D.Sc (Tech) Julius Luukko for authoritative guidance and suggestions during the project as the collaboration on the software development has been most educational, which I esteem greatly.

I would like to thank all of my friends for bearing me along these long years as I have trudged towards my accreditation. Particularly I would like acknowledge my parents for everlasting support and highly enjoyable exchange of thoughts.

In Kouvola August 6, 2009

Juha Salli

> *"Scientific knowledge is judgement about things that are universal and necessary, and the conclusions of demonstration, and all scientific knowledge, follow from first principles (for scientific knowledge involves apprehension of a rational ground)".*
>
> Aristotle, Nicomachean Ethics

Table of Contents

Abbreviations and Symbols

Symbols

e_i	Worst case execution time of a task.
m_i	A selection criterion for the RTOS
p_i	Execution period of a task.
w_i	Weight factor of a RTOS selection criterion
M	Criterion sum (fitness value)
U	CPU utilisation

Greek Prefixes

µ	Micro [$1 \cdot 10^{-6}$]

Abbreviations

µC/OS-II	MicroC/OS-II (Real-time operating system by Micrium)
x86-64	a 64-bit expansion set of the CISC x86 instruction set architecture.
A/D	Analog to Digital Conversion
AIS	Application Image Script
ANSI	The American National Standards Institute
API	Application Program Interface
ASCII	American Standard Code for Information Interchange
ASMBL	Advanced Silicon Modular Block
BRAM	Block select RAM (A base memory unit of Xilinx's FPGA)
C89	ISO/IEC 9899:1989 standard (C programming language)

C99	ISO/IEC 9899:1999 standard (C programming language)
CCS	Code Composer Studio
CISC	Complex Instruction Set Computer
CLB	Configurable Logic Block (A base logic element of Xilinx's FPGA)
COFF	Common Object File Format
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSL	Chip Support Library
D/A	Digital to Analog Conversion
DAC	Data Acquisition and Control
DBX	Debugger for Unix
DCM	Digital Clock Manager
DDD	Data Display Debugger (Open source debugging utility)
DMA	Direct Memory Access
dMAX	Dual Data Movement Accelerator
DSP	Digital Signal Processor (TMS320C6727)
DSP/BIOS	Real-time operating system by Texas Instruments
DWARF	Debugging With Attribute Record Format
EDC	Electronics Design Center
ELF	Executable and Linking Format
EM_WAIT	Asynchronous Wait Input
FDSP	Fixed Point Digital Signal Processor (TMS320F2806)
FPGA	Field Programmable Gate Array Processor (Virtex 4)

GCC	GNU Compiler Collection
GNU	GNU's Not Unix
GPIO	General Purpose Input Output
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
I^2C	Inter-Integrated Circuit
ICR	Interrupt Clear Register
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IO	Input/Output
IOB	Input/Output Block (a base structure of the FPGA)
ISO/IEC	International Organization for Standardization/International Electrotechnical Commission
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
LCEDS	Laboratory of Control Engineering and Digital Systems
LED	Light Emitting Diode
LSB	Least Significant Bit
MAC	Media Access Control
MFLOPS	Million Floating Point Operations Per Second
MIPS	Million Instruction Per Second
NMI	Non-Maskable Interrupt
OSFW	Operating System Framework
PC	Personal Computer
PDA	Personal Document Assistant

PDF	Portable Document Format (Document file format by Adobe)
PLL	Phase-Locked Loop
RAM	Random Access Memory
RMS	Rate Monotonic Scheduling Algorithm
ROM	Read Only Memory
RTF	Rich Text Format (Document file format developed by Microsoft)
RTI	Real-Time Interrupt
RTOS	Real-Time Operating System
RTS	Run-Time-Support
Rx	Reception
SARAM	single-Access Random Access Memory
SIO	Stream Input / Output
SDRAM	Synchronous Dynamic Random Access Memory
SPI	Serial Peripheral Interface
SRAM	Static Read Access Memory
SVN	Subversion (Open source version control utility)
TCB	Task Control Block
TI	Texas Instruments
Tx	Transmission
UHPI	Universal Host-Port Interface
USB	Universal Serial Bus
UTF	Unicode Transformation Format
VHDL	Virtual Hardware Description Language

1 Introduction

Digital real-time control algorithms and applications have become a more viable solution for many areas of engineering along development of sophisticated digital signal processors (DSP). As a part of that, real-time operating systems (RTOS) have gained more ground, in a wake of advancement of digital signal processor class, as a software groundwork for digital signal processing algorithms. The real-time operating system offers distinct methods to resolve design problems typically encountered in development of embedded applications. Common problems or concerns in embedded software development are priority inversion, usage of shared resources, avoidance of deadlocks, and sufficiency of system memory. Well designed real-time kernel can offer easy solutions to aforementioned challenges. From a developer point of view, the RTOS is a very tempting platform to utilise because using it solves a plethora of design traps, for instance problems with memory allocation, IO communication buffers, and error handling to name a few. Aforementioned issues can be bypassed by using a modern and versatile RTOS. Designing programs for real-time environment is convenient as the developer can focus on application creation, instead of writing supporting application programming interfaces (API). A typical commercially viable real-time kernel is very scalable, configurable, and offers wide range of services at the developer's disposal. Therefore, it's up to the developer, whether to choose an expensive, feature rich commercial RTOS kernel or enhance a modest kernel with additional code.

A downside of using the RTOS is that random access memory (RAM) and read only memory (ROM) requirements compared to ordinary round-robin type software architecture are considerable. In addition, the RTOS increases computational load (overhead) from two to four percent of total available central processing unit (CPU) resources. However, responsiveness of the RTOS will remain almost coincident regardless an application code growth, providing that application requirements aren't too strict. Real-time kernels are increasingly popular in embedded applications such as hand-held music players, cellphones, or personal document assistants (PDA) [1, p. 35–71]

1.1 Real-Time Operating Systems

A real-time operating system is a specific type of operating system, where task applications and the kernel are compiled and linked together. Commonly, the real-time kernel is a multitasking system that manages available CPU resources and internal communication services between system's tasks (threads). A scheduler (dispatcher) is an important part of the RTOS as it shares CPU time between the application tasks. Semantic importance of tasks is typically arranged by a given priority, thus context switch decisions are based on readiness and priority of tasks [1, p. 35 – 40]. The most significant difference between a non-real-time kernel, such as Linux™, is that in the RTOS environment, a starter program typically initialises the system's hardware and boots the kernel up, while in a non-real-time system the kernel is loaded into memory first, followed by separately compiled applications. The type of RTOS kernel can either be hard or soft real-time depending on task completion timing requirements. In a soft real-time system none of the system's or user's tasks need to be accomplished in a preset time frame. On the other hand, in a hard real-time system, every task needs to be computed on time without exception [2].

The kernel can either be preemptive or non-preemptive, while the latter is seldom offered by commercial vendor. Non-preemptive real-time operating systems are typically used in environments, where responsiveness is not an issue and fast interrupt handling is a desired feature. In this case, an application can use non-reentrant functions as cooperative multitasking lets only a single task at the time to access a shared resource. A preemptive operating system is typically used when response latency is the key factor. The preemptive kernel gives control of CPU for a task that is ready-to-run and has the highest priority among the tasks, thus program execution is preempted in favour of a task with the highest priority [1, p. 40–43]. The performance of software depends on available program cycles, that is CPU usage, where the kernel, user application code, additional overhead from context switches, memory routines, and interrupt handling require their share of available CPU time. Timing requirements for a real-time kernel can be examined mathematically with a rate monotonic scheduler theory, which determines doctrinaire dividing line between hard real-time and soft real-time systems.

Real-time operating systems are mostly used in embedded environments, thus efficient programming tools are required for software development and accessing low-level registers of underlaying hardware. Therefore, a typical RTOS is written in part with high abstraction level programming language, for instance C or C++ and partially with instruction set assembly language, which provides all necessary methods to manage any

registers found in a modern day microprocessor or digital signal processor. Generally, tools of the real-time operating system integrated development environment (IDE), for instance compiler, comply to software standards of International Organization for Standardization/International Electrotechnical Commission (ISO/IEC). Hence, the standard C is used extensively in embedded software development.

1.1.1 Memory Handling

Commonly, memory allocation functions of the standard C as defined by The American National Standards Institute (ANSI) and ISO/IEC are not suitable to be used in the RTOS environment. This is mainly a consequence of small amount of system memory available for the heap and non-standard compilers as not all vendors of embedded systems provide support for dynamic memory allocation in their IDE compiler. The standard C functions `malloc()`, `calloc()`, and `realloc()` use system's heap for dynamic memory reservation, which may lead to problems arisen from memory fragmention and shortage under heavy memory usage. Thereby, real-time kernels offer an alternate way to handle memory after a dynamic fashion as an memory application program interface (API) is typically provided for the purpose. Generally, any embedded memory allocation methods found in a modern RTOS are related in some degree to the memory allocator proposed by Kernighan and Ritchie [3].

1.1.2 Internal Communication

A typical, modern day RTOS offers various methods for intrinsic communication needs; a message mail box, message queue, and event flag signalling can be used to pass information between tasks. Generally, messaging services are handled after a reentrant fashion by the kernel, thus design problems concerning mutual exclusion and data integrity of communication data are mostly looked after by the RTOS. However, buffer structures needed by communication stream input and output (SIO) algorithms need to be programmed and checked to be reentrant as the RTOS manages only data transfer between data containers.

1.1.3 Periodic Program Execution

Periodic delays provided by the RTOS are typically based on utilisation of hardware timers, where a task is suspended for preset tick count and shifted back to scheduler's ready list after the delay is expired. The method is applicable in applications, wherein program execution does not need to be deterministic as significant fluctuation can be observed in a typical delay implementations. However, deterministic timing can be achieved by utilising semaphores with a separate hardware timer interrupt, but the solution is most likely bound to the utilised hardware and not very portable [4].

1.2 Theoretical Premises

This study is a complementary part for the active magnet bearing (AMB) control software development project, which was introduced by the Laboratory of Control Engineering and Digital Systems (LCEDS) to examine possibilities of DSP based control software. One of the goals concerning software development for the DSP board by Electronic Design Centre (EDC-DSP), was to conceive a real-time operating system environment, where a single software framework solution could prepare the way for a broad spectrum of divergent applications. The main focus of the thesis is to research; characteristics of real-time operating systems, an idea of semantic framework working on top of the utilised RTOS, and to find solution how a real-time operating system framework environment can be used with the EDC/DSP board.

There is not only one widely accepted theory how a real-time operating system should be designed as the real world requirements and restrictions establish boundaries for any embedded system designs. The subject of the study is to find solutions for the research problems in close relation to the underlying real-time operating system. Thus, certain approved theories and axioms concerning embedded system design are examined. Notional basis of the study is the theory of Rate Monotonic Scheduling (RMS), proposed by Liu and Layland, which dictates doctrinaire limits for a hard real-time operating system. The RMS theory alleges that an optimal scheduling between a periodic task set with a preemptive scheduler can be achieved when a task with shorter period is given higher priority [5]. The theory is studied to find, whether will the OSFW comply to preset timing requirements.

The field of research of embedded programming is known to harbour several permissive estimations regarding embedded system software design, which are more conceptual and subjective than well founded theories. Two informal design paradigms; top-down process decomposition and structured analysis are used in combination to explicate and evaluate requirements and solutions with relation to design of the OSFW. The process decomposition methodology is used to contemplate semantic abstraction levels from requirements to actual code and to profile development steps in accordance to requirements and goals. The method is utilised to model principled structures and interfaces of the OSFW.

The structured analysis is a procedural development resource and quite similar to the process decomposition methodology. The main difference between the two is that approach of the structured analysis rests on procedural programming language semantics. The method produces detailed software building blocks, for instance flow charts, thus, it's utilised in detailed software development as a programming implement. Aforementioned development paradigms are applied to design problems concerning reliability, performance, interoperability, maintainability, portability, and modularity issues. Performance of forged software is examined with the RMS theory.

1.3 Development Issues

Generally, a research project, for example control application, that utilises a real-time operating system, have to settle to available services and characteristics found in the selected RTOS, unless a comparative study is carried out to select the most suitable RTOS. Therefore, finding a suitable RTOS can be difficult, although all modern real-time operating system offers similar services for general use. Hence, real-time operating systems, which are sophisticated enough to provide support for applications that require hard real-time deterministic execution in periodic manner are rare. The purpose of the research is to develop a real-time kernel based operating system framework API that enhances the RTOS in a way that application development can be done with little or no knowledge of an underlying operating system. Concerning the study, interesting development questions are:

- Availability of services and application interfaces of the selected RTOS.
- Dependencies and restrictions concerning the hardware and software.

- Availability and usefulness of the chip support libraries (CSL) of the DSP.
- Methodology of hardware abstraction layers (HAL).
- Real-time computing issues concerning the system's performance and timing requirements of deterministic, periodically computed control algorithms.
- Safety measures concerning the system's hardware and software errors.
- Development of communication driver routines for data exchange between CPUs of the EDC-DSP board and problems concerning internal data exchange between system APIs.
- Dynamic memory allocation issues concerning internal and external memories.
- Memory access protection, e.g. RAM and SDRAM residing program code, buffers, and dynamic memory allocation issues concerning internal and external memories.
- Structural composition of skeletal software structure that allows increment of external software components.
- Documentation issues concerning upkeep and further development of the software in multi-developer environment.

The aim of the software development is to come upon a solution that provides sufficient universally applicable application program interfaces and system services for variety of the EDC-DSP board based applications. The OSFW is to be a skeletal software groundwork, in which different kind of software components, for instance libraries or extensions, can be added seamlessly.

The OSFW fills the gap between general use real-time operating systems, for example the MicroC/OS-II^{TM} (µC/OS-II), and hardware specific real-time operating system such as DSP/BIOS^{TM}. Commonly, this kind of universal, highly device independent service layers are not found in most real-time operating systems. Therefore, a need for such real-time operating system is considerable especially in academic research projects, where the underlying system must be revisable, code-wise transparent, and fully documented.

1.4 Outline of the Research

The thesis is composed of a literature study, an analysis of the requirements, a synthesis of the real-time operating system framework, programming of the software, testing of the system, and documentation and literary work as a report of the thesis. The keynotes of the study are contemplated to discover what kind of a real-time operating system could be used, how exact requirements for the system could be fulfilled, and how the produced software conforms to theoretical premises and software development standards. A semantic operating system framework is modeled and programmed as proposition for found development issues. The software is tested in the real world hardware environment and findings of the system's performance are construed in theoretical context. In this thesis it's shown how:

- A real-time operating system is enhanced and expanded to follow through exacting requirements of a complex hard real-time control system application.
- A semantic software framework provides higher and transparent abstraction level services typically found in non-real-time operating systems.
- A modular program structure assists maintenance and in-depth documentation promotes effectual continuation of development.

Chapter 2 describes the functioning and composition of the EDC-DSP board's hardware, especially the TMS320C6727 digital signal processor, which is the target environment for the OSFW. In Chapter 3 is reported how the requirements are analysed, what design methods are used, how the real-time operating system is selected, how the architecture of the OSFW is designed, and how the application program interfaces of the OSFW are modelled. Chapter 4 shows how documentation of the OSFW is arranged, how the system is programmed, how separately developed source codes are assembled and tested. In Chapter 5 is examined how the OSFW operates in theoretical context as the system's CPU usage and memory consumption are benchmarked. In addition, the congruence of the OSFW software between the project's objectives is analysed. Chapter 6 discusses the conclusions and future prospects concerning the OSFW.

2 Control System Hardware

2.1 EDC-DSP Control Board

The EDC-DSP board (Fig. 2.1) is a prototype of versatile data acquisition and control (DAC) system designed by Electronic Design Centre of Lappeenranta University of Technology. The board is designed to suit wide spectrum of divergent digital control and data collection applications as it's composed of two separate circuit boards. The base of the board supplies power and serves as data input and output (IO) cross-connection point for a primary board. The primary board is the effectual data processing unit and compounded of three processing units and volatile and non-volatile external memories. Therefore, the base is a standing part of the EDC-DSP board as the primary board can be changed, if needed as different application may require novel board schemas. Concurrent model of the base board contains a power unit, cross-connection slots, and light emitting diodes (LED) for signalling purposes. The LEDs are commonly used for software debugging and the system's state indication providing that an application utilises the LED outputs in the field programmable gate array processor (FPGA).

The current incarnation of primary board is meant for heavy DAC applications, thus it's equipped with two high performance processors; a field programmable gate array processor of Virtex-4^{TM} family by Xilinx, and a 32-bit floating-point digital signal processor: TMS320C6727 by Texas Instruments (TI). Third processor on the board is a mediocre capacity fixed point digital signal processor (FDSP) TMS320F2806, also provided by TI. The FDSP's role is to act as bootloader for the high performance processors and command relaying gateway between user interface, for instance a terminal, and the operating system framework.

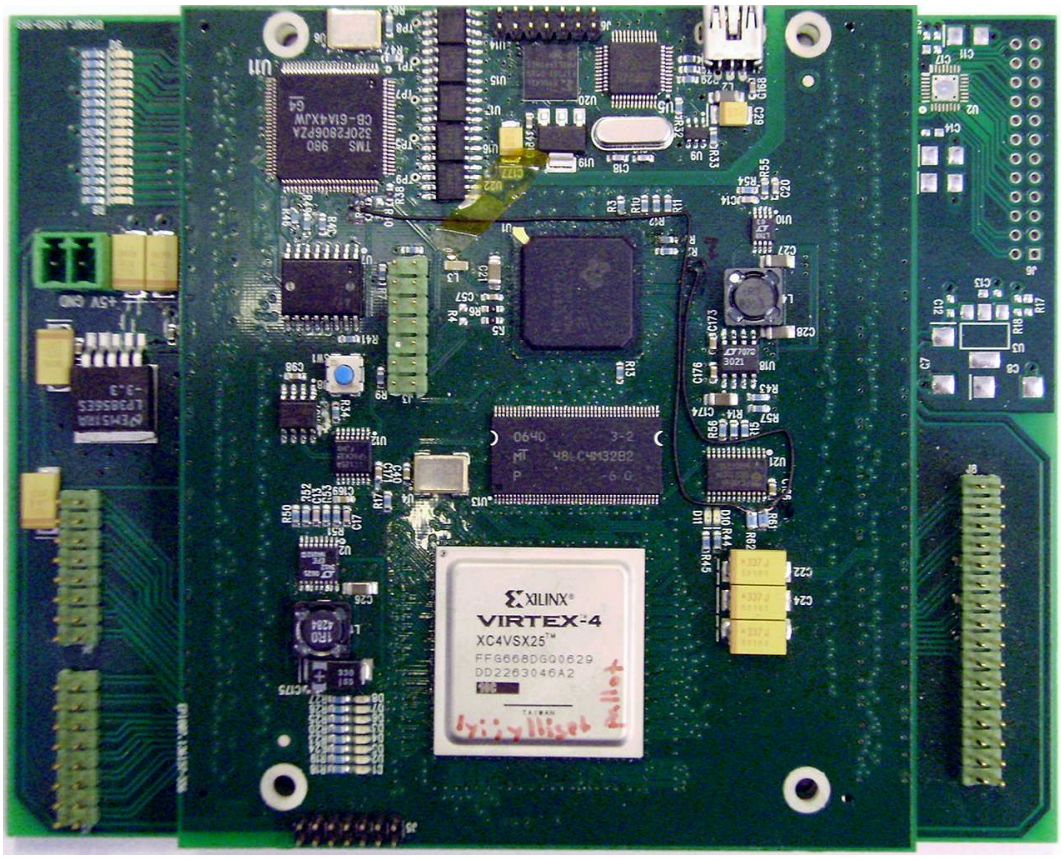

Figure 2.1: The EDC-DSP Multipurpose Data Acquisition and Control Board with a general-purpose IO board underneath.

2.1.1 Internal CPU Connections

The DSP and FDSP are in-lined with two hardware serial communication lines; inter-integrated circuit (I^2C) and serial peripheral interface (SPI), to make external boot up procedures possible for both the FPGA and DSP. The serial communication controllers within TI's DSPs are united in way that either I^2C or SPI serial busses can be used at the time. Currently the FDSP utilises the I^2C for booting as the DSP's chip suffers from silicon flaws [6], which prevents usage of the SPI hardware for external boot up procedure. Therefore, the I^2C serial controller is currently used as boot up interface. The FPGA and FDSP are connected via general purpose IO pins, which used to boot the FPGA. 32-bit address lines of the FPGA and DSP are interconnected and multiplexed with the external synchronous dynamic random access memory (SDRAM) to provide memory mapped databus between the processors. In addition, the FPGA can be used as an external memory providing that the FPGA is programmed to emulate memory types supported by the DSP's external memory interface (EMIF)

In-lining of address pins ensures that data path's width is wide enough for demanding DAC algorithms, where the FPGA collects data, passes it to the DSP, which manages CPU intense calculations and returns calculated data back to the FPGA. In order to ensure synchronised data exchange, a dedicated signal line is connected from the FPGA to DSP's asynchronous wait input of the EMIF controller (EM_WAIT), which manifests as a non-maskable interrupt (NMI) in the DSP.

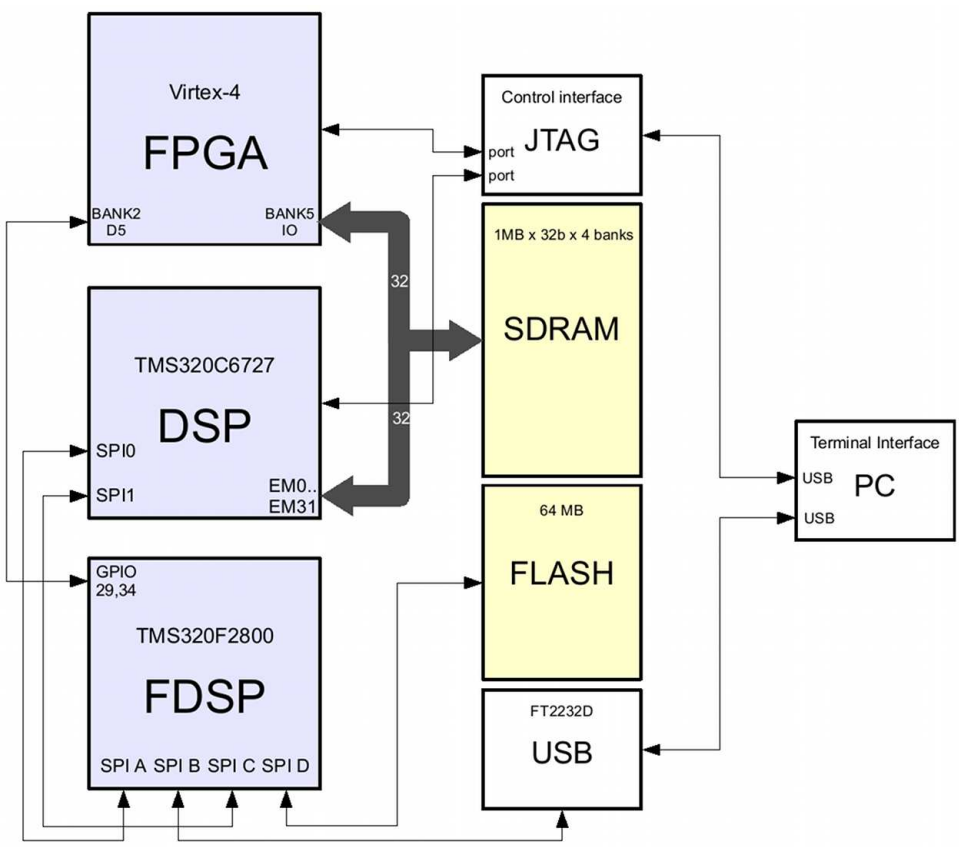

Figure 2.2: A schematic layout of the EDC-DSP board's memory and data bus connections.

2.1.2 External Memories

The primary board's adequacy for DAC applications is ascertained with augmented external memory as 4 MB SDRAM and 64 MB flash memories are incorporated on the board. the SDRAM is in-lined with address pins of the DSP, hence it's mapped directly into the DSP's address space. The flash memory is in-lined with the FDSP and serves as boot image cache, where several FPGA and OSFW application image files can be stored. The OSFW utilises SDRAM extensively as all memory intensive activities are performed in SDRAM, for example dynamic memory allocation.

2.1.3 External Data Connections

The primary board is fitted with a universal serial bus (USB) control chip to provide interconnectivity between a personal computer (PC) and the FDSP. At the present, the USB connection is used to load program images of the DSP and FPGA applications into the external flash memory. The FDSP is equipped with an internal flash residing control software that is used to boot both the DSP and FPGA via the USB interface. The FDSP utilises the flash memory with a basic file system that supports program image download into a specified memory address. However, the file system's failings prevent individual image removal. A DOS™ environment program (SFProg) [7] is developed to function as PC interface for the EDC-DSP board. The current version of SFProg makes it possible to load application images into the flash memory, erase the memory, inquire status of the program execution of the DSP and FPGA, and pass commands after a limited fashion. The FDSP's software and SFProg are under development, thus the FDSP's capability to pass commands to the DSP at current state is practically non-existent. The software of the FDSP and SFProg application will be revised to provide better interconnectivity in future revisions. A joint test action group (JTAG) standard connection is provided to enhance the primary board's debugging capabilities with any JTAG supporting integrated development interfaces. The JTAG interface is in-lined between the DSP and FDSP, thus granting access into CPU registers of the DSP and FDSP. Generally, the JTAG is used for software development in debugging phase as applications can be downloaded and monitored via the interface. A schematic description of the EDC-DSP board's external connections is presented in Figure 2.2.

2.2 TMS320C6727 Digital Signal Processor

TMS320C6727 is a high performance digital signal processor that can achieve execution rate of 2400 million instructions per second (MIPS) or alternatively 1800 million floating point operations per second (MFLOPS) at the maximum operating speed 300 MHz. The peak performance is achieved by carrying out eight instructions in parallel on each clock cycle, where six commands can be floating point instructions. The DSP's internal 256KB RAM and 384KB ROM memories are connected to CPU via an on-chip memory controller, thus there's no factious separation between program code memory and data memory addresses.

The memory controller makes it possible to fetch an instruction, access data in internal memory, and perform an operation in peripheral address space in parallel within one clock cycle. Furhermore, fast program cache prevents, but not necessary eliminates, instruction fetch misses and execution delays originated from pipeline bubbles [8, p. 2]. The memory controller's throughput enables program execution from external memory, thus auxiliary SDRAM can be utilised as program memory for large applications or library routines.

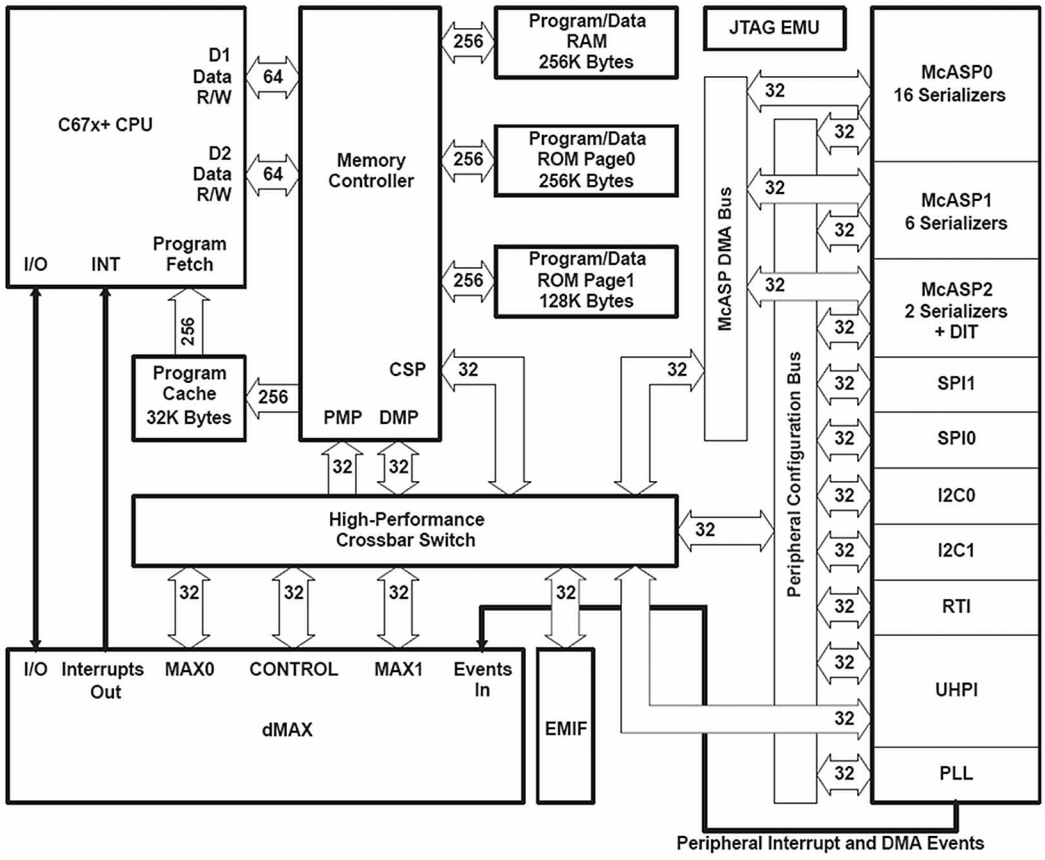

Figure 2.3: Block Diagram of the TMS320C6727 digital signal processor [8, Fig. 1-1, p. 5]. © Texas Instruments

2.2.1 Consequential Properties

The DSP is equipped with a central data path cross-connection bridge: the Crossbar Switch (Fig.2.3), which manages data flow between CPU, dual data movement accelerator (dMAX), and universal host-port interface (UHPI), and various system peripherals, for instance memory. However, cross-connection is not fully extensive as the UHPI cannot access peripheral devices via this route. The crossbar moves data after a parallel fashion proving that other bus masters are not trying to access the same peripheral coincidentally

[8, p. 2–5]. Concerning the OSFW, most essential services provided by the DSP's on-chip accessories are real-time interrupts (RTI), direct memory access via dMAX, external memory interface, and serial connections via the SPI and I^2C

2.2.1.1 Real-Time Interrupt Module

The RTI module provides a digital watchdog and deterministic general-purpose counters for various timing applications, for instance RTOS heartbeat tick. The watchdog makes it possible to control the system, if erroneous program execution or hardware malfunction crashes the system, thus preventing program code execution. Therefore, upon system failure, the watchdog initiates an interrupt (INT0) that resets the system providing that the watchdog counter and interrupt are enabled. The OSFW utilises the watchdog counter interrupt as a safety measure and interrupt of the RTI timer as a tick clock for the RTOS. The RTI is configured via memory mapped control registers. Figure 2.4 presents a schema of the digital watchdog [9, p. 10–12,14].

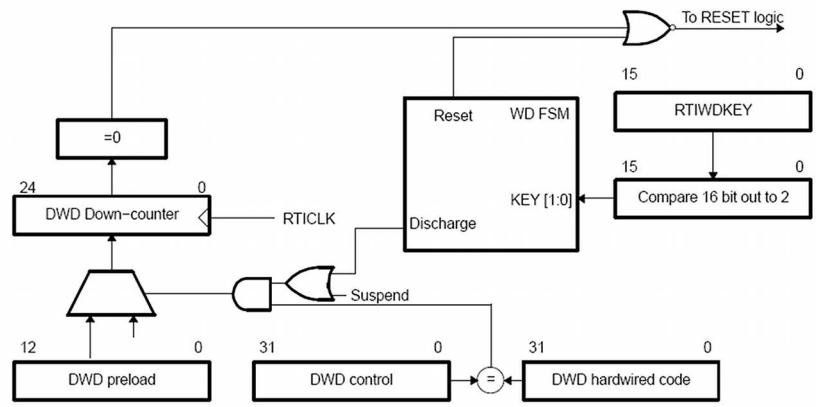

Figure 2.4: Operation Diagram of the Digital Watchdog [9, Fig. 1-2, p. 13]. © Texas Instruments

2.2.1.2 Dual Data Movement Accelerator

dMAX controller manages user-programmed direct memory access (DMA) data transfers between internal memory controller and peripherals of the DSP. The DMA routines are confined to a fixed set of DMA types; general data transfer up to a third dimension, First-In-First-Out (FIFO) write and read transfers, one-dimensional burst transfer, and SPI slave transfer. The usage of dMAX is bound to utilisation of DMA transfer entry

tables, wherein all possible DMA transfer types are configured, for example SPI slave transfer used by the OSFW. The Control registers of dMAX are memory mapped, thus configuration of all DMA events is done in centralised nexus [10, p. 73].

The controller notifies CPU via dedicated interrupt lines (INT7, INT8), whether for example FIFO status or end-of-transfer DMA events need to be addressed. These interrupts originate from dMAX's channel synchronisation events, which can occur concurrently. Therefore, dMAX utilises prioritised order of event processing, which is divided into high and low priority classes. The controller can simultaneously employ one event from each priority classes, thus the highest priority event of each class can be processed at once as all events belong inherently to one or the other [10, p. 23–72].

2.2.1.3 External Memory Interface Controller

The EMIF controller provides memory mapped connection with supported external memory devices, such as single data rate SDRAM, asynchronous flash, and static read access memory (SRAM). Therefore, external memory devices can be accessed with byte or word addressing as internal memory. The controller provides a non-maskable interrupt (EM_WAIT), which is used to provide asynchronous setup delay for external flash memory. However, the EDC-DSP board utilises this interrupt as a synchronisation signal between the FPGA and DSP. The controller is configurable via CPU memory mapped registers. [11].

2.3 TMS320F2806 Digital Signal Processor

TMS320F2806 is the fixed point DSP based on high performance static complementary metal oxide semiconductor (CMOS) technology that yields 100 / 60 MHz operating frequencies with low power consumption. The FDSP supports software development with C or C++ high-level programming languages, thus making it suitable for general RTOS computations. Even though the processor is fixed point, the 32x32-bit multiply-accumulate capability with 64-bit execution feature enables the FDSP to compute higher resolution mathematical algorithms. The core incorporates eight-level-deep protected pipeline with in-lined memory access, thus providing high execution rate without falling back on high-performance memory.

In addition, the boot ROM incorporates standard tables, such as sinusoidal wavetable, suitable for mathematical algorithms. The processor provides JTAG boundary scan support, firmware protection with 128-bit security key, CPU timers and a watchdog module, serial port peripherals (SPI, I^2C), general purpose IO pins (GPIO), and on-chip flash and single-access RAM (SARAM) [12, p. 11, 36–42].

2.4 Virtex-4 FPGA Processor

The Virtex-4 is high-performance field programmable gate array processor family, which is composed of three divergent application solution models; logic, digital signal processing, and full featured embedded platform. The processor is based on combining advanced silicon modular block (ASMBL^{TM}) architecture, wherein a general routing matrix provides an array of routing switches for each component. Therefore, every programmable block or element is cross-connected to the switch-matrix. Architectural high-lights are [13]:

- Programmable IO blocks (IOB) form an interface between GPIO's and internal programmable logic.
- Configurable logic blocks (CLB) offer combinatorial and synchronous logic.
- 18KB dual-port block RAM (BRAM) modules offer FIFO capability and can be cascaded to form larger memory blocks.
- Digital clock manager (DCM) blocks offer clock source; multiplication, division, distribution delay compensation, and phase-shifting.
- Digital signal processing block can be utilised to execute typical signal processing needs as the block embodies; 18x18-bit multipliers, integrated adder and 48-bit accumulator.

The vendor offers wide selection of third party intellectual property (IP) core blocks for commonly used complex functions and virtual devices including; PowerPC^{TM} processors, Ethernet media access controllers (MAC), serial transmitter-receivers, specialised DSP blocks, digital and phase-matched clock management, and source synchronous interface blocks. Concerning the OSFW, availability of IP blocks makes the FPGA a very versatile data acquisition platform.

3 Design of the RTOS Framework

The development of OSFW is carried out by a composite of structured analysis and high-level functional specification paradigms, thus not resorting on any formal methods such as finite state machine or state charts. These informal paradigms were chosen as they could depict the real-time operating system framework in satisfactory detail. Despite the fact that formal methods are preferred in development of systems where execution substantiation is imperative as suggested by Laplante, benefits derived from the formal methods concerning the OSFW are arguable. Furhermore, no formal method can prove a real-time system to be correct, error free, and mission critically safe [14, p. 167–168]. Therefore, the selection and study of underlying RTOS and its complementary framework is emphasised.

The design of OSFW software is started by establishing premises and requirements of the study, hardware and software induced constraints, desired functionality of the system, and goals of the framework environment development. Hence, wide set of requirement specifications are explicated to create a solid ground for software development. A preliminary study is conducted onto preset standards as the research is compounded of literary work that concentrates on timing requirements, specification of software components, quality and security issues, architectural perspectives of the OSFW's application programming interfaces, the RTOS selection, availability and usability of chip support libraries, usefulness and practicability of existing FDSP software and its APIs, target DSP's hardware functioning, and user interface issues.

The elicited requirements are formulated after a similar fashion to the *"Recommended Practice for Software Requirements Specification"* standard by Institute of Electrical and Electronics Engineers (IEEE) [15]. Hence, detailed analysis of the requirements established the course for software development and set the scene for selection of programming paradigms.

3.1 Analysis of Software Requirements

The dissection of collected requirements is carried out after a comparative fashion, where demands and available hardware resources are constantly appraised. The analysis is founded on hardware data sheet literature sources and experience gained from the EDC-DSP board prototypes in earlier research projects. As a result of former projects, capacity of the EDC-DSP board is designed to be excessive and most suitable for taxing embedded applications. Therefore, generality of standards for the OSFW are set as principles, thus speculative validation is not needed as the hardware's capacity either complies to the requirements or not. Concerning the OSFW software, performance demands are bound to resources of the hardware.

3.1.1 Hardware Functionality

The EDC-DSP board's functionality is considered to be suitable to fulfill all requirements concerning the OSFW development. However, operation semantics of the board were examined to find, if there was any limiting factors concerning RTOS utilisation and software development. The review concentrated on issues such as: IO connections, sufficiency of system's memory, data busses, the real-world capacity of the DSP, cross-connection requirements between the DSP and FPGA, and interfacing with PC via the USB. Commonly, nothing conclusive emerged that would prevent using the EDC-DSP board as foundation for the OSFW.

3.1.2 Dynamic Memory Allocation

Dynamic memory allocation in the RTOS applications requires special attention as available memory for a system's heap block is typically very small. In addition, extensive dynamic allocation may fracture the heap, thus possibly leading to a software failure as required memory space may not be available because of fragmentation. Thus, the most troublesome issue concerning a real-time system application with dynamic memory allocation is to specify correct memory size as most memory allocators found in RTOSes handle memory allocation by hard coded memory blocks [1, p. 274]. In the OSFW, dynamic memory allocation is designed to utilise variously sized memory blocks in external SDRAM via the RTOS memory management services.

3.1.3 Timing Issues

The most significant requirement regarding the OSFW is hard real-time performance with the worst case timing demand: 100 µs. This is a consequence of high-speed digital control algorithms, for instance AMB control, which are to execute in kHz domain after a periodic fashion. The standard is exacting as the OSFW must keep up with a control algorithm, which designates that the real-time operating system's tick clock must be at least equal to the worst case time window requirement. To emphasise the importance of the requirement, the tick clock frequency used in the OSFW exceeds a typical RTOS setup by factor 10-100 [1, p. 68]. Furthermore, additional CPU overhead derived from excessive tick interrupt with scheduler execution and diverse task switching, degrades overall available CPU time. The more demanding timing window is, the greater percentage of available CPU resources are forfeited in internal RTOS operations, for instance context switch. Experience gathered from past studies have revealed that high speed tick rates with real-time operating systems are feasible as presented by Penttinen [16].

CPU utilisation is the sum of time fragments of individual tasks, where it's assumed that every task is periodic and holds unique execution period. The sum is calculated as [14, p. 11]

$$U = \sum_{i=1}^{n} \frac{e_i}{p_i} \tag{3.1}$$

where *U* is total CPU utilisation sum of tasks, *n* is number of individual tasks, e_i denotes worst case execution time of a task, and p_i designates the execution period of a task. The equation shows that CPU load value exposes scheduling problem, even if a single task exceeds its time frame, thus indicating that the task is not rate monotonic schedulable. This is very undesireable as uncertain execution compromises deterministic behavior of the system. CPU utilisation is a crucial criterion concerning the hard real-time execution as it represents time of non-idle processing. Therefore, measured available CPU processing time reveals the system's risk of being overburden.

The RMS theory states that any set of *n* periodic tasks is rate monotonic schedulable if the total CPU utilisation is

$$U \leq n(2^{n^{-1}} - 1) \tag{3.2}$$

The theoretical limit for a set of *n* periodic tasks is found when *n* is let to approach infinity

$$\lim_{n \to \infty} n(2^{n^{-1}} - 1) \tag{3.3}$$

Thus, by modifying the function (3.3)

$$\lim_{n\to\infty} n(2^{n^{-1}} - 1) = \lim_{n\to\infty} \frac{2^{n^{-1}} - 1}{n^{-1}} \tag{3.4}$$

and by utilisation of a basic derivate formula [17, p. 134]

$$\frac{d}{dx}a^x = a^x \ln a \tag{3.5}$$

And the rule of l'Hôpital [17, p. 130] applied to the equation (3.4)

$$\lim_{n\to\infty} \frac{2^{n^{-1}} - 1}{n^{-1}} = \lim_{n\to\infty} \frac{\ln 2(2^{n^{-1}})(-n^{-2})}{-n^{-2}} \tag{3.6}$$

And by pursuing the standard limit [17, p. 131]

$$\lim_{n\to\infty} x^{n^{-1}} = 1 \tag{3.7}$$

The equation (3.6) takes a shape

$$\lim_{n\to\infty} \ln 2 \approx 0.693 \tag{3.8}$$

Thus, the sum of CPU utilisation of individual tasks must satisfy the requirement

$$\sum_{i=1}^{n} \frac{e_i}{p_i} \leq 0.69 \tag{3.9}$$

for the system's periodic tasks to be schedulable in accordance to the RMS theory.

Table 3.1: CPU Utilisation zones [14, p. 11]

Utilisation [%]	Zone Type
0–25	Extremely safe
26–50	Very safe
51–68	Safe
69	Theoretical limit (RMS Theory)
70–82	Questionable
83–99	Dangerous
100+	Overload

The RMS theory alleges that if the system's CPU utilisation is under the theoretical limit, the system is guaranteed to be rate monotonic schedulable. Even though the theory sets a borderline for scheduling of periodic tasks, the system's tasks may well be rate monotonic schedulable with CPU utilisation rate higher than theoretical maximum [14, p. 96]. In a Table 3.1 is presented a classification of CPU utilisation zones as proposed by Laplante.

3.1.4 Additional Software Components

The target CPU environment of the OSFW is TMS320C6272 digital signal processor, which is interconnected with the FDSP via SPI and FPGA via general address pins in-lining. The requirement of CPU interconnectivity requires that a detailed interface between other processing units must be provided. Therefore, the main issue is, how data transfer via the FDSP and memory usage with the FPGA's flash memory can be arranged? The solution doesn't differ much from casual API planning as the design paradigms, used in the OSFW development, inherently produces desired interfaces within the DSP's software. Further development concerning the FPGA and FDSP was left open as it outreaches scope of application of the thesis.

3.1.5 FPGA Data Interface

The requirement for a data exchange interface between the FPGA and DSP expects that the DSP can be utilised as a computational asset as the FPGA is the functional interface for power electronic devices. Commonly, the FPGA is employed to perform rapid analog to digital (A/D) and digital to analog (D/A) conversions as acquired raw data is processed in the DSP. This is the case for a typical application that needs the DSP's resources. Memory addressing pins of the FPGA and DSP are in-lined, thus providing memory mapped access for shared resources. Synchronisation between the processors is a considerable challenge as the DSP's doesn't provide any external interrupt pins [8, p. 44]. Therefore, the FPGA is in-lined with DSP's EMIF controller device to utilise non-maskable interrupt; EM_WAIT. The signal is used to provide syncronisation event for any control applications as data exchange between the FPGA and DSP needs to occur on time. The FPGA will be programmed to provide designated memory blocks in future revisions, thus a driver interface is designed to function as transceiver that reads data from the FPGA's memory, processes it and puts output data back into the FPGA.

3.1.6 Quality of Program Code

Demands concerning the quality of software are expressed in three domains: program code must be comprehensively documented, used software tools ought to support maintainability and development in multi-developer environment, and functions of the OSFW must be reentrant. The requirements are considered to be model guidelines in the OSFW development. Documentation is not merely a description about how a program works but an actual development tool, which can be used to trace changes and seek design errors. The concept of traceability depicts relations between the system's requirements and design, thus a documentation of a program code is traceable, if it's done conscientiously along software development as stated by Laplante [14, p. 235]:

> *"Traceability is particularly important in real-time systems because often design and coding decisions are made to satisfy hardware constraints that may not be easily associated with a requirement. Failure to provide a traceable path from such decisions through the requirements can lead to difficulties in extending and maintaining the system".*

Pervasive documentation that is started in the beginning of software development will accomplish the requirement. Concerning the OSFW, the paradigm of software development is that documentation must provide extensive estimation about the system's: APIs, semantic layers, functions, objects, macros, detailed usage, solutions concerning encountered hindrance, for instance hardware bugs, and divergences found in the real-world solution and it's literature counterpart. Generally, documentation issues are very important as without extensive details of code semantics and structures, maintainability and further development becomes more difficult. Writing a commentary for code is an arduous task, thus a third party utility tool: Doxygen [18] is used for the assignment. Software tools for integrated development are typically provided by a hardware vendor, thus most of development and testing is typically done in an utilised IDE. However, code portions that are hardware independent, for instance high abstraction level APIs, can be developed in alternate environment providing that compliance to standards of the main IDE tool is ensured. This widens possibilities and reduces dependence on a single development tool as embedded systems are well known for being difficult to test. Maintainability issues are typically related to code structures of software, for instance source code arrangements or declaration semantics. Therefore, an IDE or separate documentation tool should be able to represent structures of software in cohesive manner, for instance as charts.

Multi-developer environments are not broadly supported in hardware vendors' development tools. Hence, dedicated version control development tools are commonly used for the task. Concerning source code arrangements, it's adviceable to hold to ASCII or unicode transformation format (UTF) file formats in source code files and avoid proprietary ones, if possible. This ensures maintainability and interoperability as standard file formats are generally supported and accessible from development tools in different operating system environments. The target DSP of the EDC-DSP board dictates testing development environment: Code Composer Studio® (CCS), which is utilised in the OSFW's testing phase.

3.1.6.1 Reentrant Program Code

A basic paradigm for an embedded software is reentrancy, thus every function of the OSFW is designed to fill the demand by utilising local variables, multiple access exclusion, and contex switch denial where applicable. In consequence, utilisation of global variables, as available-for-all type, is generally avoided, thus most global objects are either bound to certain task or API and are typically used as a data buffer objects. Usage of generic buffer arrays is arranged in a way that only defined set of service layer functions or a task can access the resource, where access properties are secured by the accessing function.

3.1.7 Security Considerations

The EDC-DSP board is meant solely for scientific use, thus security considerations and problems are not emphasised in the study. If the board is to be commercialised, a full security inspection should be conducted on the device to discover potential security threats and to find solutions for efficient immaterial rights protection. In general, security threats emerge in two semantic domains as both the hardware and software flaws can put the system at risk. The most significant hardware and software threats occur, when the system's physical integrity is compromised and data bus traffic and communication interfaces are exposed to an invader. It's vital to acknowledge security hazards by mapping out a security architecture that incorporates both the hardware and software designs for every aspect how the system is used. The security architecture should comply to following guidelines [19]:

- Software interfaces should make use of efficient and powerful fallback with a strong encryption as data is to remain enciphered between the device and a remote access terminal.
- Datagrams can be encrypted efficiently with special type encryption processors.
- Algorithms, procedures and connections between different APIs must be designed to identify and reject erroneous data streams and login attempts.
- Security architecture should employ various well tested and well-known scrambling methods.
- All unsafe accessories and data connections that are used for a system development should be eliminated in a commercial grade device.

Physical attacks can be controlled when a system is fitted with security chip along safety encapsulation, which purpose is to perceive physical intrusion passed into the system. The safety chip can perform many tasks upon detected attack varying from program data deletion to raising an alarm. However, to shield a system against an accomplished assailant is difficult and expensive.Therefore, the level of protection must be designed in accordance to data and immaterial rights and intended environment of use.

Software based attacks usually try to exploit weaknesses in system APIs, thus every interface of the system must use encryption keys, which are calculated and transferred in encrypted manner between a terminal and the system. In addition, well protected system should enforce encryption key invalidation, for example after three failed login attempts, or alternatively enter into a dormant, unoperational state that requires maintenance from the system's supplier [20].

3.1.8 User Interface Requirements

Concerning the EDC-DSP board's CPU in-lining, interfacing with the DSP is a considerable challenge. Communication information needs to flow via a PC terminal interface through the FDSP into the DSP. Therefore, communication design is carried out only with the DSP as interface development for the FDSP is out of the scope of this study. Basicly, the interface needs to provide means to pass commands into the DSP, thus the interface at the DSP end is designed to parse and process commands which are formed as strings.

3.2 Applied Design Paradigms

The top-down process decomposition design methodology paradigm is employed in a form of V design method, The methodology is used to control the project starting from requirements gathering and advancing into programming of the software and ending up to testing of the OSFW. The preliminary study, which focused on software semantics, revealed early on that the framework is to be an application program interface extension for an existing real-time operating system. Therefore, it would have been excessive to utilise formal methods as the actual embedded system is the underlying RTOS.

Concerning the OSFW, all essential data that is needed to laid out the framework's construction is collected punctiliously. This data includes subjects that concern functionality of the hardware, external interfaces of the FDSP and FPGA, performance of the RTOS and API functions, hardware constraints, software characteristics, that is reliability, and maintainability and portability of the software. Aforementioned issues can be classified as measurable and semantic functionality, where the code's functionality and performance can be surveyed, while semantic features remain subjective and are not observable via software execution. [14, p. 164]

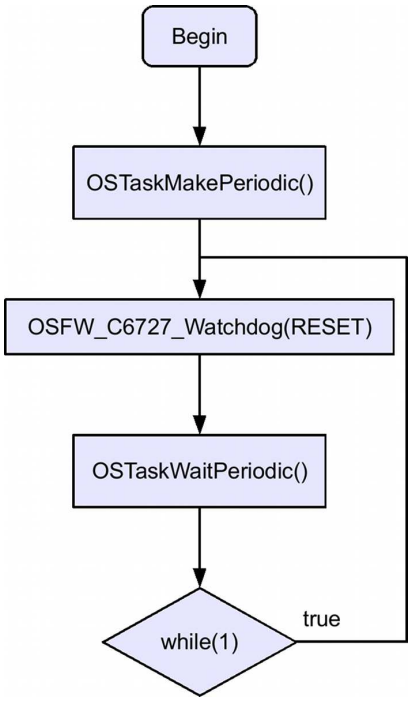

Figure 3.1: Flow chart of the OSFW's watchdog task.

The high-level function analysis is utilised in form of flow diagrams to represent all required functions and service layers. Development of the flow charts advanced in parallel with the structured analysis as the first designed software structure was data objects (structures), which passes through application programming interfaces of the OSFW. The interfaces are designed to be flexible, thus composition of data objects can be revised as the software evolves. The design process of the system's task and function modelling availed from flow charts as arguments, return value, and internal code operations of a function could be delineated in a precise format. Therefore, all tasks and functions, excluding simple macros, are drawn up as flow diagrams. Basicly, substantive programming is done in this phase of development. As an example, Figure 3.1 presents a flow chart of the OSFW's watchdog task.

3.2.1 V Design Method

Software structures of the OSFW and code testing phases are designed in accordance to design paradigm called the V -method. The aim of method is provide systematic development convention that advances steadily after a step-by-step fashion from top to bottom, where for each decision making step an additional testing plan generation is required. One of the strong areas of the V -design method is that it begins with requirement specification and includes distinctive design steps all the way to actual end product. Hence, the paradigm requires that hardware boundaries and requirements for software must correlate, whether a mapped out system is feasible. If contemplation of these requirements is well founded, effectual software development can proceed to designing of the software architecture.

In the OSFW context, the software architecture designates a conceptual design, where all segments and transition interfaces from higher level abstraction APIs to low-level drivers of the software are depicted narrowly. The last step, prior starting actual code writing, is to decide conventions of the program code including; usage of pointers, design of data structures and unions, unified error handling, classification of functions, utilisation of external program sources, and programming styles and documentation issues. Thereby, this design method yields a layered semantic structure, where on the top are design premises and in the bottom the results of the design that is the program code. The design work flow of the OSFW software and testing practices is presented in Figure 3.2.

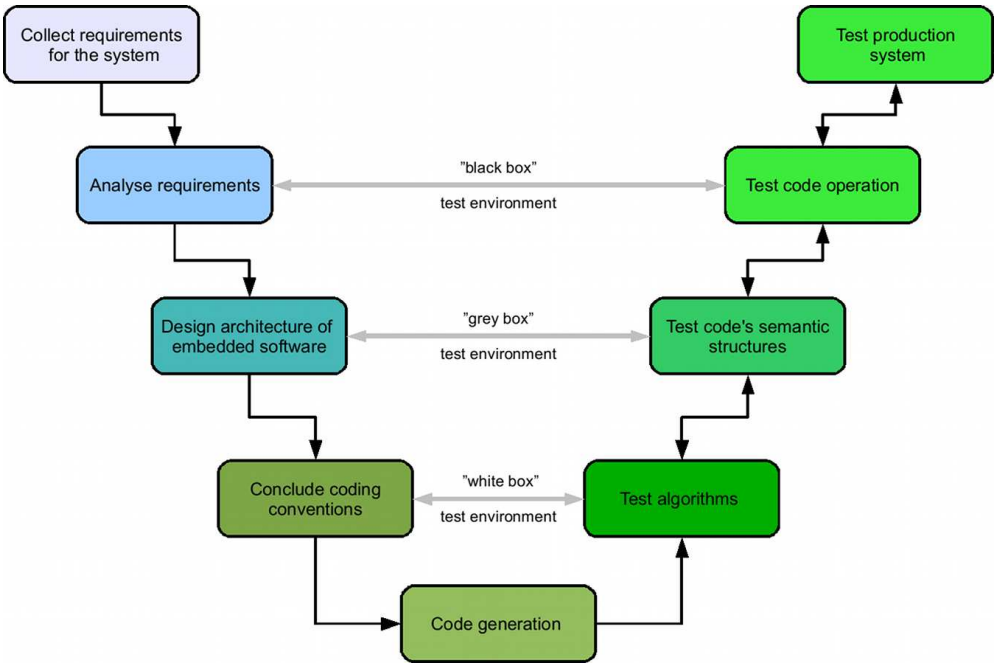

Figure 3.2: Semantic structure of the V- design method used in the OSFW software design.

3.2.2 Modularity Semantics

One of the generally accepted software development principles used with the OSFW software development is the Parnas Partitioning, also referred as the Information Hiding principle, introduced by Parnas [21]. The methodology states that an algorithm, which reconstitutes as a function or module, should be generalised, independent of other modules, and possess narrowly predefined outline. This means that a function is programmed to execute only what is absolutely necessary concerning the function's purpose. The main motivator to program algorithms in this way is to avoid complexities and logical errors induced by having inordinate amount of features in a single function. Furthermore, the design paradigm's position is to produce general use software, thus interfaces within the code are considered to be fundamental and semantically fixed.

Generally, code structures are discerned as modules, in which interaction between the modules occurs via application program interfaces, thus internal functioning of a module is invisible to other modules. Therefore, this programming paradigm produces considerably high cohesion modular software structure, which is maintainable and robust. All functions of the OSFW are designed in accordance to the principle, which effected natural boundary between high abstraction level APIs and low-level service layer functions, for instance class drivers.

3.2.3 Coding Conventions

Programming styles and means have evolved along development of programming language semantics and are normally bound to the used programming methodology, for instance differences in coding conventions between procedural and object-oriented programming languages. However, subjective variance of programming styles exists within the syntax of programming languages, wherein perspectives of different applications are presented, for instance disparity between hard real-time embedded and ordinary graphical user interface (GUI) programming. Hence, subjective programming conventions concerning software development of the OSFW are:

- Clarity of code's syntax and readability are highly prioritised as Doxygen managed documentation is embedded in source code, in which descriptive commentary of an object, a declaration, and a function is placed before actual code.
- Additional elucidating commentary is placed in complex code portions, where mere syntax won't necessarily reveal inner functioning of code.
- Program code of the OSFW is divided into several modules; API, hardware initialisation, interrupt service routines (ISR), and system tasks to improve maintainability. Hence, the separation is extended into source code and header files as the modules are placed in distinctively labeled source code files.
- User applications can be programmed separately and merged with the OSFW easily.

The OSFW is programmed in accordance to the ISO/IEC 9899:1989 (C89) standard owing to limitations of Code Composer Studio IDE, as the C89 is the latest standard supported by the CCS's compiler. Analogy with the standard is achieved by utilising compiler options after a pedantic fashion in all phases of programming.

3.2.3.1 Naming of Functions and Variables

The naming convention of the OSFW is designed to be self-explanatory and similar to distinctive naming practices of µC/OS-II. The naming model is composed in a way that acronym of the real-time operating system framework is positioned in the beginning of function's or variable's name in capital letters, thus making the OSFW's code distinguishable. In addition, the naming style also prevents double labeling issues, for instance, if

an auxiliary library have a function with the same name as the OSFW's function. After the framework label is a section that depicts a purpose followed by qualifier, which describes a function. However, the naming convention is considered as guideline, thus small alternation between the purpose and qualifier labels appears when applicable.

For example, the naming semantics of a ringbuffer's initialisation function differs from a function of word register bit manipulation [22, p. 45–46]. This naming convention applies also to all global variables, type definitions, enumeration types, static and dynamic defines, and objects of the OSFW. However, local variables that are declared within a function are labelled freely. In order to improve code's readability all function arguments, which are pointer, are denoted with a letter p placed in front of the argument's name. This notation style is used also with variable elements of an object to avoid misapprehension and programming errors, thus integer, real, and boolean variables types are differentiated with letters n, f, and b respectively.

3.2.3.2 Data Objects

The primary study showed that efficient data transfer through the system's APIs would benefit from services provided by the RTOS. Therefore, all transferable data is packed up into interface specific data objects, thus making it possible to pass vast amount of data via RTOS queues as pointers. The data structures designed for the OSFW are semantically objects, but they lack substitution principle features of object-oriented programming. Thus, objects of the OSFW should be considered as a variable sized, external memory residing data containers. In addition, parameterized dynamic memory block sizes of various API buffers dictates the maximum usable size of a data object.

3.2.3.3 Pointers

In order to achieve high software performance, it's usually undesired to move any argumental data from internal or extended memories into a task's stack. Additional overhead caused by a single function's arguments copy is irrelevant and does not take many program cycles, but in an large embedded system with limited resources, it's most certainly an issue. Furthermore, filling the stack with objects need more allocated memory, which in turn may not be possible in memory challenged systems. Therefore, many memory related restrictions can be solved by using memory address pointers instead of actual data,

for instance void type addressing. Using pointers is highly efficient, but also prone to errors, unless exhaustive white box testing is discharged as programming with pointers is the known Achilles heel of procedural programming languages, for example the C language. The OSFW is programmed in accordance with basic principle that a function ought to get its arguments and pass return value as pointers when applicable.

3.2.3.4 Error Handling

General error handling isn't a principle feature concerning the OSFW's operational functioning. However, an extensive error handling interface is designed to manage system wide error events. The interface is intended to be used as an optional part in user applications to help software development and to pass error information from user written functions. The OSFW utilises the error handling API only in the standard C substitutes, memory checking function and in the system tasks, excluding the error handler task.

3.2.3.5 Compiler Directives

The compiler directives are defined constants, which are used as compiler's preprocessor directives to make it possible to compile the OSFW after a conditional fashion. Thus, software's versatility increases as different code portions can be compiled depending on needs of a target application. The source code of the OSFW is enhanced with several compiler directives, which affects to system's memory handling, functionality of the error API, system tasks, variable initialisation, and system testing. The set of compiler directives is placed in a single header file to promote practicability. The directives are presented on Appendix VI, 1.

3.2.3.6 Avoidance of Shared Data Problems

The problem of shared data is taken into consideration by designing the OSFW architecture to use global variables as little as possible, to utilise local variables extensively, to form a mutual exclusion mechanism into shareable data objects, and to take advantage of data transferring services of the RTOS, for example queues. In addition, designed buffer structures and associated functions promote data integrity as all buffer related data objects are stored in dynamically allocated memory blocks in secure manner.

3.3 RTOS environment selection

Two real-time operating systems; MicroC/OS-II by Micrium and DSP/BIOS by Texas Instruments are selected as proprietary candidates to be used as a groundwork for the real-time operating system framework. These particular candidates were chosen as they both incorporate required characteristics to fit the bill in accordance to the demanding real-time control system standards. Furthermore, µC/OS-II has a long history in academic world and mission critical applications, whereas DSP/BIOS is an approved and commercially accepted realtime operating system. Concerning the OSFW software development, the Code Composer Studio IDE provided by Texas Instruments can be used regardless the chosen RTOS as the OSFW's target DSP is made by the vendor. The premise of selection methodology is to chart and appraise the key features, strong areas, and failings of the both candidates.

3.3.1 Attributes of DSP/BIOS

DSP/BIOS is a light-weight, deterministic and preemptive real-time operating system, which incorporates a modular and selectable software component structure. The software is an integral part of the CCS development environment, which promotes application development endeavour as development tools provided by the CCS are geared towards DSP/BIOS. In addition, extensive documentation is provided for application development, real-time debugging, and API programming. Close integration with the CCS is also frailty of DSP/BIOS, as Texas Instruments has designed the RTOS to be used in its own DSP products, which makes it practically impossible to port a DSP/BIOS based application into an alternate DSP platform. Furthermore, application program interfaces of DSP/BIOS are specific for TI's DSP products. The key properties of DSP/BIOS are:

- Availability of software based interrupts.
- Data-pipes management in form of SIO drivers is provides with data stream IO structures.
- Benchmarking support and comprehensive chip support library, which provides low-level driver layers for various system [23].
- Priority sharing between tasks, thus enabling round-robin scheduling.

- Graphical user interface with versatile debugging tools.
- Application development avails from tools provided by Code Composer Studio IDE.

One of the shortcomings of DSP/BIOS is that a number of available priority levels is only fifteen [24], although a priority is shareable between tasks. Therefore, problems may arise in large CPU intensive applications in a form of increasing overhead jitter, if many tasks occupy the same priority level. Licensing policy of DSP/BIOS allows a free runtime licence, but the development environment licence must be purchased, which is inconvenient in academic development projects.

3.3.2 Attributes of MicroC/OS-II

The MicroC/OS-II is highly portable, deterministic, and preemptive multipurpose real-time operating system applicable to be used in safety-critical environments [1, p. XV]. Well designed portability ensures that µC/OS-II is not tied into any particular software or hardware development environment, thus it can be utilised with any IDE or development tools which provides C89 (ANSI C) standard compiler. Additionally, target hardware's port assembly code is also required. µC/OS-II can be used in variety of CPU architectures ranging from 8-bit miniature systems up to 32-bit floating point digital signal processors, thus making the kernel inherently hindered with 8 and 16-bit restrictions, for instance maximum number of tasks is 63 of all available 64 priorities [1, p. 79]. The key assets of µC/OS-II are:

- Functioning of the kernel is proved to be reentrant and fault tolerant.
- Software architecture of the kernel is composed of three semantic layers; processor independent, processor dependent, and application specific. The architecture makes it feasible to port the software from one hardware architecture to another.
- The scheduler induced jitter in task context switch is minimal as priority sharing is not allowed, thus making the round-robin type scheduling impossible.
- Priorities can be handled after a dynamic fashion as the scheduler can elevate a task's priority to avoid priority inversion. Furthermore, the amount of priority levels can be controlled to save memory.

- API layer for dynamic memory handling is provided.
- Tasks stack sizes are adjustable and a software tool is provided for run-time stack size calculation.
- Services provided by the kernel are coded after a systematic fashion and can be parameterized via compiler directives.
- External features can be added via low level hook functions [1, p. 287–336].
- Memory optimization can be achieved with cross-use of the kernel services [1, p. 244,271].
- Licensing µC/OS-II for academic purpose is free [1, p. 567].
- The kernel can be examined thoroughly as the documentation is a book: *MicroC/OS-II The Real-Time Kernel*.

The code of µC/OS-II is highly portable and processor-specific program code is generally written in assembly. Therefore, optimum CPU specific machine code is not necessarily derived as processor-specific code is typically written by a user. The reason for non-optimum code generation is typically found in an insufficient compiler as undocumented CPU quirks, for instance silicon errors, register tweaks, that are not hardcoded into the compiler, may alter the code performance considerably. Another consequence of portability is that µC/OS-II lacks all IO application interfaces, device drivers, and general debugging tools. Therefore, all system services and application program interfaces, which are to provide aforementioned features, must be programmed by a developer.

3.3.3 Selection Criteria

The selection criteria methodology used to select the RTOS is a calculation a weighted sum of thirteen evaluation characteristics; Interrupt latency, thread management, memory requirements, scheduling mechanism, internal communication possibilities, after-sale support, application availability, portability, source code's availability, functional issues such as context switch time, software price, availability of software development platforms, and support of network protocols as suggested by Laplante [14, p. 134–139]. These criteria are weighted and evaluated in accordance to the requirements of the OSFW by the author.

The sum of selected criteria is calculated with an equation [14, p. 137]:

$$M = \sum_{i=1}^{13} w_i m_i \tag{3.10}$$

Where $M \in [0, 13]$ denotes the weighted sum of selected criteria as metric fitness value, $w_i \in [0, 1]$ is a weighting factor, and $m_i \in [0, 1]$ is a selection criterion. Both of the RTOS candidates provide a typical set of characteristics found in a modern, preemptive real-time operating system. However, it's noticeable that neither of the kernels provide a memory garbage collector as memory management service within the memory management APIs. One of the main strengths of DSP/BIOS is the chip support library, but using it may not be effortless as API structures of the CSL might be fundamentally changed by the software vendor, for example distinction of the chip support libraries between versions 2 and 3 [25]. Commonly, the greatest asset of µC/OS-II, portability, is also the most arduous feature as all IO related services and APIs must be programmed as part of an application.

Table 3.2: Decision table of OSFW's real-time control system selection.

Criterion	Specification	w_i	DSP/BIOS [m_i]	µC/OS-II [m_i]
m_1	Minimum Interrupt latency	1.00	1.00	0.95
m_2	Task management	0.75	0.60	0.50
m_3	Memory requirements	0.80	0.90	1.00
m_4	Scheduling mechanism	0.50	0.75	0.50
m_5	Intertask communication methods	1.00	1.00	1.00
m_6	Product life cycle support	0.50	1.00	0.60
m_7	Availability of supporting software	1.00	1.00	0.50
m_8	Hardware compatibility	1.00	0.10	1.00
m_9	Availability of source code	1.00	0.25	1.00
m_{10}	Context switch time	0.90	1.00	1.00
m_{11}	Price of software	0.75	0.50	1.00
m_{12}	Availability of development platforms	1.00	0.10	0.90
m_{13}	Supported network protocols	0.25	0.50	0.00
M	Fitness metric		6.90	8.83

The fitness metric M shows that the MicroC/OS-II real-time operating system is more appropriate for the OSFW development than DSP/BIOS. Generally, both of the candidates answers to the requirements but µC/OS-II is chosen resulting from better portability, licensing policy, and being independent from development environment and mercantile software development tools. A thorough calculation of the fitness metric values is presented on Appendix I, 1

3.4 Development of the OSFW Architecture

Development of software architecture for the OSFW is a logical consequence of analysis of the system's requirements, which draws the guidelines for the system's functionality and needed robustness. The key characteristics of the architecture are RTOS integration with the hardware, interconnection between the FPGA and DSP, user interface, memory handling and memory maps for SDRAM and RAM, system APIs, hardware abstraction layer semantics, software extensions for the µC/OS-II, and periodic task execution. Generally, the architecture is an information hiding API that is divided into several operational interfaces, which are designed to be independent as much as possible.

Concerning the information hiding principle, the OSFW is designed as a two-level hierarchy model, where a hardware driver level manages hardware operations (hardware and class drivers) and higher API level provides abstracted interface to the system's resources. Generally, the line between the abstraction layers is tractable as the API semantics allows virtually unrestricted access to low level drivers, thus ensuring transparency and maintainability.

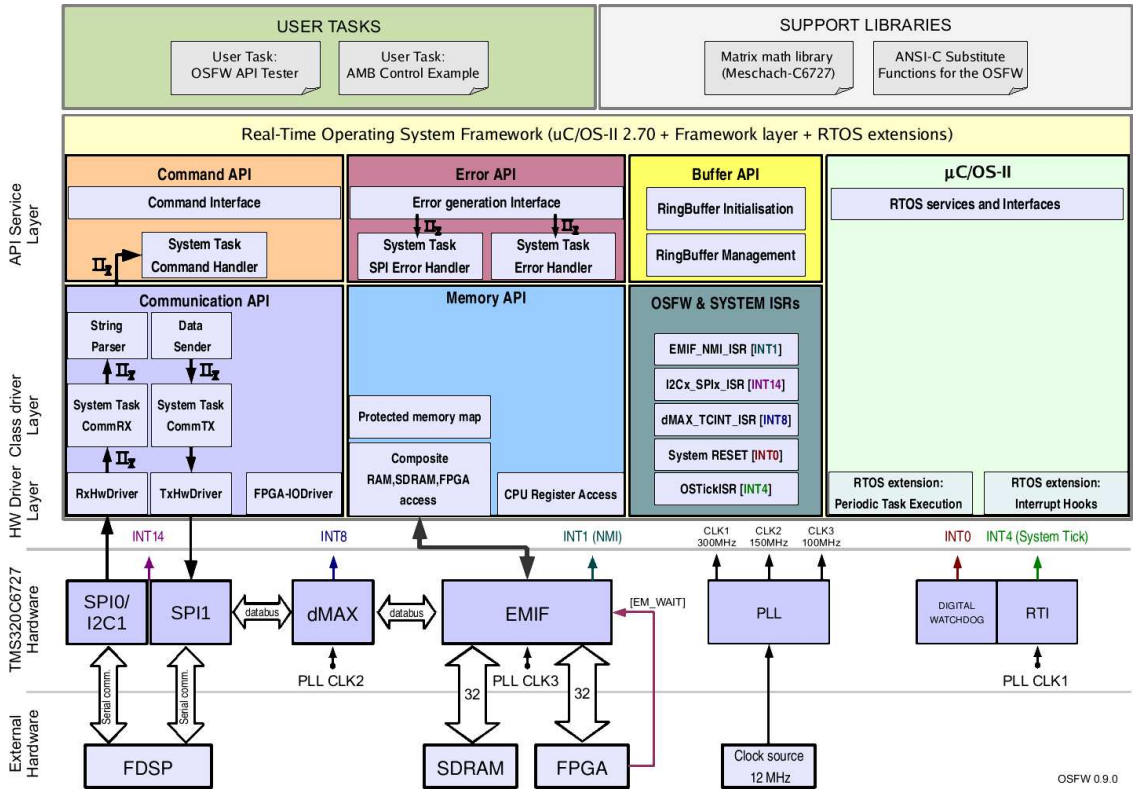

Figure 3.3: A detailed layout of the OSFW's application interfaces and interaction with the EDC-DSP board's hardware

In addition, expanded generality of abstraction layers provides an open development environment as appointed interfaces between the modules ensures interoperability regardless of alteration of internal functions of the OSFW. Therefore, reusability of code is increased as the API modules can be enhanced instead of complete rewrite. In Figure 3.3 is presented the OSFW's combined software and hardware layout.

3.4.1 RTOS and Hardware Integration

The OSFW's integration with µC/OS-II is materialised in two different abstraction levels. The lowest point of interaction occurs at interrupt service routine layer, where hook functions are called from the ISR functions of µC/OS-II. The hook functions are separated out from assembly code of the RTOS port, thus only hook functions calls need to be added in the processor specific assembly program code. Hence, portability and maintainability is ensured as the source code of the OSFW and µC/OS-II are kept apart as much as possible.

The most considerable utilisation of various µC/OS-II services comes true in the API layer of the OSFW as all data communication between the APIs is carried out via the queue service layer. Hence, the most utilised RTOS services are; dynamic memory handling, message queues, and mutex semaphores. Although the programming interfaces of the OSFW rests mainly on services of µC/OS-II, the OSFW's APIs incorporates high cohesion modularity, meaning that the underlying RTOS services can be changed effortlessly, for instance porting the OSFW to a different RTOS environment. Major effort is made to solve the integration issues to come up with flexible and versatile API structure, thus reducing reliance to the underlying operating system.

3.4.1.1 CSL Utilisation

All hardware related information is defined within the OSFW's header files as no additional chip support library is utilised caused by complexities found in implementation of the CSL of Texas Instruments. Generally, utilising the CSL would have created an additional abstraction layer, thus making the software unnecessary complex and difficult to maintain. In addition, the CSL is geared towards the DSP/BIOS operating system, thus using it would have hindered the project's endeavour to ensure portability and indepence of proprietary software.

3.4.1.2 MicroC/OS-II Port for TMS320C6727

The MicroC/OS-II is designed to be highly portable, thus programming a port for novel processor architecture is convenient. The processor-specific code is divided into three source code files; *OS_CPU.H*, *OS_CPU_C.C*, and *OS_CPU_A.ASM*. *OS_CPU.H* contains processor-specific definitions for data types, stack operation (growth) direction, and hardware registers. Extension hooks and stack initialisation function are placed in *OS_CPU_C.C* source code file.

The OSFW's extension hook functions are not included in *OS_CPU_C.C*, but in it's own extension source code files; *osfw_os_ext.c* and *osfw_os_ext.h*. This is due to inherited programming practice of the periodic extension, where additional hook functions are separated from *OS_CPU_C.C*. The most crucial system functions are written in assembly and gathered in a separate assembly language file *OS_CPU_A.ASM*, which incorporates functions for context and task switch, system start, and system's tick interrupt service routine.

Porting µC/OS-II requires that the target processor is supported with a compiler that is capable to produce reentrant code and the processor incorporates interrupts and general purpose timers that can used as a system clock. In addition, all interrupts must be manageable with the C programming language. Generally, hardware stack must be supported and instruction set of the processor must include commands to load and store CPU registers and the stack pointer [1, p. 287].

The port of µC/OS-II for TMS320C6727 digital signal processor is collaborative work brought forth by Ming Zeng, Kenneth Blake, and Julius Luukko, wherein only low level extension hooks `OSEnterCriticalHook()` and `OSExitCriticalHook()` are added during the OSFW project.

3.4.1.3 Data types

The port file (OS_CPU.H) [1, p. 291] includes redefinition of C89 data identifier types with additional names to enhance readability and to emphasise memory size manifestation of a variable in the RTOS program code. The OSFW utilises these enhanced data type names to promote interoperability with µC/OS-II as presented in Table 3.3

Table 3.3: Specific data type definitions of µC/OS-II.

Standard definition	µC/OS-II definition	Explanation
unsigned char	Boolean	Only the LSB is used
unsigned char	INT8U	Unsigned 8 bit integer
signed char	INT8S	Signed 8 bit integer
unsigned short	INT16U	Unsigned 16 bit integer
signed short	INT16S	Signed 16 bit integer
unsigned int	INT32U	Unsigned 32 bit integer
signed int	INT32S	Signed 32 bit integer
float	FP32	Single precision real number
double	FP64	Double precision real number

3.4.1.4 MicroC/OS-II Configuration

Services of µC/OS-II are utilised extensively as; event flags, message mailboxes, memory management, mutual exclusion semaphores, and message queues are all enabled and configured. Thus, all essential operating system services provided by µC/OS-II are included and utilised by the OSFW. The configuration header file is presented on Appendix II, 1

3.4.1.5 Kernel Versions

The MicroC/OS-II is an evolving RTOS, thus the software's bugs gets fixed and new features are added by Micrium as the software evolves. Therefore, the OSFW is bind to quite recent major release: 2.70, even though more recent major release version 2.86 is available. The main reason for using this version is that the periodic extension, used in the OSFW, is written for and tested with the kernel version 2.70. Therefore, to avoid unwanted software components re-write, the kernel version 2.70 got selected for the framework development. However, the OSFW's source code includes preliminary adaptation of the newest µC/OS-II kernel version 2.86.

3.4.1.6 ISO/IEC Standard Functions Substitutes

In order to avoid Code Composer Studio's compiler related restrictions, functionality of the ISO/IEC 9899:1999 (C99) standard library functions `printf()` and `strtok_r()` are emulated in their embedded equivalents. The compiler of CCS, conforms only to C89

standard [26, p. 124], thus making it unable to provide support for the C99 standard token function. However, the functionality of these particular functions is needed, thus, new versions of the C99 functions are programmed as replacements. The substitutes tailored for the OSFW are:

- `INT32S OSFW_printf (const char *str, ...)`
- `void* OSFW_strtok_r (char *pStr, char *pDelim, char *pTkn, char *pSave)`

The algorithms of enhanced functions are designed to emulate their standard C library counterparts flawlessly, thus using them, from a developers point of view, is almost identical compared to the standard functions. Another motivation is to overcome software errors and inconsistent behaviour found in the standard library functions, which in turn, may have serious effects on program execution in the EDC-DSP system.

3.4.2 API Semantics

Application program interfaces of the OSFW are designed to offer express information hiding boundary between the OSFW and user applications. The system APIs are formed as a collection of semantically related functions, which collectively manages certain higher abstraction services. Generally, the OSFW's program interfaces are not interconnected, but the communication API is designed to act as data traffic nexus for all other APIs. All data flow is designed to occur in stream IO manner, where data is always kept separated and deposited in unique memory location and handled in protected mode. In addition, the OSFW's API interfacing is designed to be suggestive of µC/OS-II's API usage, thus promoting the OSFW's cohesion with the underlying RTOS. A detailed diagram of the OSFW's application program interface is presented in Figure 3.3.

3.4.2.1 Hardware Abstraction Layers

The OSFW's hardware abstraction is carried out by forming three divergent layers; API, class drivers, and hardware drivers, in which a higher level layer utilises a lower one's services, thus enhancing the abstraction of the calling layer. This methodology brings in transparent hardware abstraction grading, wherein complex register level information is

hidden but because of the hardware abstraction layer semantics, all details of the system are readily accessible. The HAL is not a rigid structure as all the OSFW's drivers or hardware accessing functions can be utilised directly. Therefore, the classification between aforementioned layers is not hard and fast but distinctly semantic.

The hardware is hidden by defining all required register and memory addresses, control and status bits, and enumerating utilised hardware events, for example interrupts. This basic hardware concealing is used mainly by low level drivers, which operates the system's hardware. Commonly, the hardware is accessed only by dedicated drivers and the system's hardware set up functions during boot up procedure.

3.4.2.2 Class and Hardware Drivers

A class driver is an appellation for drivers that do not access the system's devices directly but can control behaviour of the hardware drivers. Typically, the OSFW's class drivers are programmed to function with a dedicated hardware driver to avoid complex code and ensure maintainability. Therefore, data is transferred almost wholly after an one-way fashion from a class driver to a hardware driver or vice versa. A class driver gets data either from higher abstraction level functions or hardware driver depending on the driver's purpose. Commonly, data is transformed from an API object into data stream (string), while it passes the class driver abstraction layer, for instance a command string is parsed and an command object is formed after the parsed string.

Concerning the OSFW, all hardware accessing functions are considered to be hardware drivers. The drivers of this class are dedicated to serve a certain device, for instance SPI hardware, thus making a driver hardware specific. Generally, a hardware driver is called either by an event, interrupt, or a class driver, whereby registers of a target device are accessed by the driver. Depending on the driver's function; it either gets data from the device's transfer register and passes collected data onwards to an associated class driver or writes data to the device's transfer register and prepares the device for a sequential cycle, for example in typical send and receive operations.

3.4.3 Memory Management

The OSFW's memory handling is composed of a linker's configuration file memory allocation and dynamic run-time memory management. Commonly, RAM memory is reserved only for program and user application code. However, one megabyte of SDRAM is reserved for the OSFW, application code and additional libraries, thus the linker's configuration file contains memory location definition to move program code into SDRAM memory during external boot up procedure. The OSFW is designed to utilise memory management services of µC/OS-II to provide dynamic memory allocation capability. Typically, memory allocation functions of the standard C uses the CPU supported heap for the purpose. However, available amount of RAM memory to be used as heap may not be sufficient in demanding applications, which perform tasking mathematical algorithms such as matrix resizing. Therefore, all temporary memory, which is needed for dynamic allocation by the OSFW, is reserved from SDRAM. The memory allocator of µC/OS-II handles memory as linked lists, wherein a list item's block size is predefined and cannot be changed during run time. Dynamic SDRAM memory pool is divided into sections of various block sizes to reflect possible needs of divergent applications. A basic, 128 byte block memory allocation pool is enabled by default. Additional larger pools can be enabled by a compiler directive.

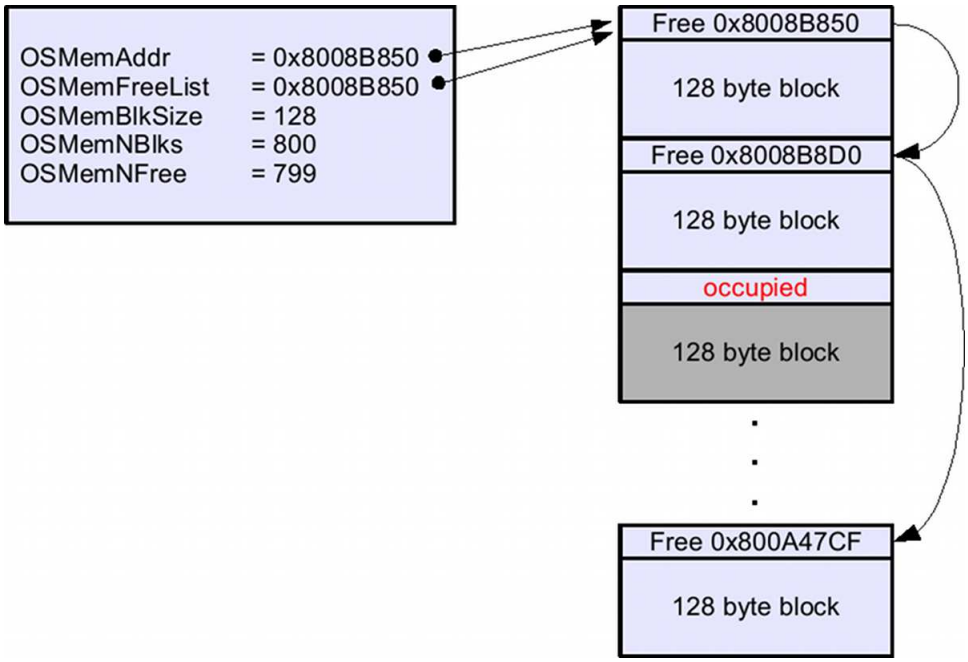

Figure 3.4: A detailed scheme of the OSFW's dynamic memory management; example of the OSFW's default dynamic memory pool [1, p. 279]

3.4.3.1 RAM Memory Map

The internal RAM memory is divided into different sections, where some memory is permanently reserved for the DSP's own purposes, for instance boot up procedure. The program code of the OSFW can be deposited into RAM or SDRAM, thus preparing the way for large applications. RAM memory allocation is managed by the CCS's linker and configured with the linker's configuration file as shown in Appendix III, 1. In Table 3.4 is presented RAM memory map used by the OSFW.

Table 3.4: The memory map of internal RAM and SDRAM in accordance to linker configuration of the OSFW version 0.9.0

Memory Block	Access	Address [Hex]	Size	Definition
iROM_BOOTLOADER	r	0x00000000	128 KB	Bootloader code
iROM_DSPLIB	r	0x00020000	48 KB	DSP library
iROM_FastRTSLIB	r	0x0002C000	16 KB	Fast RST library
iROM_DSPBIOS	r	0x00030000	192 KB	DSP BIOS
iRAM_BOOTLOADER	r/w	0x10000000	4 KB	Bootloader
iRAM_DSPBIOS	r/w	0x10001000	2.75 KB	DSP/BIOS
iRAM_FastRTS	r/w	0x10001B00	256 B	Fast RTS
iRAM_OSFW	r/w	0x10001C00	150 KB	OSFW and user applications
iRAM_FREE	r/w	0x10027400	99 KB	Free memory
SDRAM_OSFW	r/w	0x802EE000	1 MB	OSFW and user applications

3.4.3.2 SDRAM Memory Map

In the OSFW context, the memory map depicts how the external SDRAM is allocated and segmented for various memory usage requirements. The principle of the OSFW software development is to utilise SDRAM memory as much as possible and leave internal RAM for program code and additional parts of code that require fast interoperability with CPU. The system's external SDRAM memory is allocated and segmented after a semantic fashion to provide flexible and revisionable memory map for the OSFW. The composition of memory map for SDRAM is presented in Table 3.5.

Table 3.5: The memory map of external SDRAM in accordance to definitions of the OSFW version 0.9.0

Memory Block	Start Address [Hex]	End Address [Hex]	Size
FPGA data area	0x80000000	0x8000004F	80 B
Task data area	0x80000050	0x8000104F	4 KB
Communication buffer (Tx)	0x80001050	0x8000144F	1 KB
Communication buffer (Rx)	0x80001450	0x0000184F	1 KB
Error buffer	0x80001850	0x80001C4F	1 KB
Command buffer	0x80001C50	0x8000204F	1 KB
Data buffer	0x80002050	0x8007F04F	500 KB
Ring buffer object array	0x8007F050	0x8008B84F	50 KB
Dynamic mem.blkc: 128 B	0x8008B850	0x800A484F	100 KB
Dynamic mem.blkc: 256 B	0x8008A850	0x800BD84F	100 KB
Dynamic mem.blkc: 512 B	0x800BD850	0x800D684F	100 KB
Dynamic mem.blkc: 1 KB	0x800D6850	0x800EF84F	100 KB
Dynamic mem.blkc: 2 KB	0x800EF850	0x8010884F	100 KB
Dynamic mem.blkc: 4 KB	0x80108850	0x8012184F	100 KB
Dynamic mem.blkc: 8 KB	0x80121850	0x8015384F	200 KB
Dynamic mem.blkc: 16 KB	0x80153850	0x801B784F	400 KB
Free memory area	0x801B7850	0x802EDC4F	1,241 KB
Null space	0x802EDC50	0x802EDFFF	0.92 KB
RTOS program area	0x802EE000	0x803E7FFF	1 MB

3.4.4 FPGA Interface

The analysis of the DSP-FPGA interconnection requirement didn't incorporate any exact details how wide the data path should be. Therefore, an IO data exchange interface and a low level driver are designed to make it possible to transfer data between the DSP and FPGA in versatile manner. The data exchange interface is a pair of composite IO objects; *DSP_Data* and *FPGA_Data*. The IO object's elements are pointers to actual variable array structures; *OSFW_DSP_OUT*, *OSFW_DSP_IN*, *OSFW_FPGA_IN*, and *OSFW_FPGA_OUT*, which holds the data.

The current version of OSFW supports data transfer of ten independent IO variables between the FPGA and DSP. The size limit isn't fixed as the IO variable array structures can be resized providing that the memory map of SDRAM is updated in accordance to sizes of the IO objects or alternatively the *DSP_Data object* is kept in RAM memory or failing that, in free memory space of SDRAM. The data objects are basicly memory addressing frames, which read and write data in accordance to the pointed structures, thus making it possible to utilise the driver to provide data transmutation.

Memory allocation of the *DSP_Data* object is selectable, thus either RAM or SDRAM memory space can be allocated for the object depending on a compiler directive. Therefore, potential need for high speed data transmission capability is implemented as CPU can address RAM residing IO object up to three times faster than SDRAM located object. However, the capacity of the EDC-DSP board fulfills the principle requirements, even thought the IO object is retained in SDRAM.

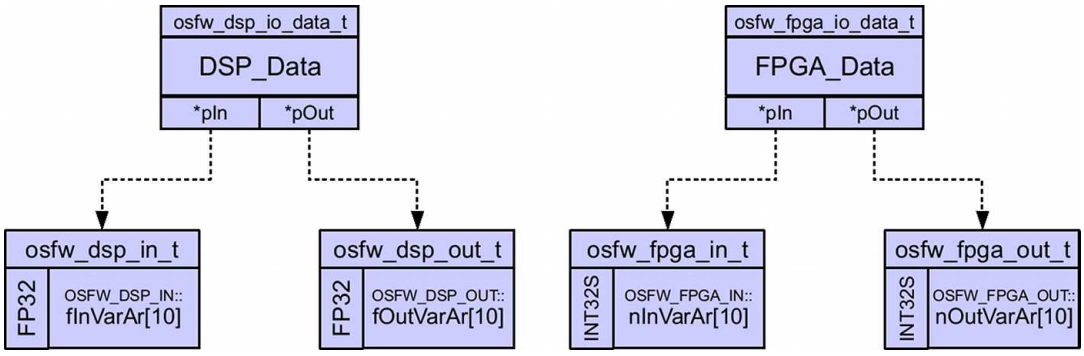

Figure 3.5: Schematic diagrams of the OSFW's FPGA interface IO objects.

The FPGA IO driver [22, p. 62] is designed to handle data in arrays as presented in Figure 3.5 and to execute data conversions from a variable type to another in accordance to driver's control object's parameters. The driver performs all necessary data format transformations, which is scaling and changing an integer to float and the other way round in different phases of a control algorithm's cycle. Therefore, the driver is basicly a half-duplex IO accessing conversion routine, which reads data from a source address, converts it in accordance to the function's arguments and writes output back to a destination address.

Synchronization of separate CPUs is a testing problem, thus the scope of the thesis isn't aimed to generation of detailed solution for CPU synchronisation. However, a simple solution for the FPGA's sampling cycle signalling is proposed, as the data transfer between the FPGA and DSP requires strict synchronizing mechanism, because of the DSP based control algorithms. The NMI interrupt event is designed to be used in two ways; continuously or as one-off synchronization signal.

The type of synchronisation is selected with a compiler directive, which shapes the interrupt service routine of the NMI in accordance to the selection. If the NMI event is chosen to come about continuously, the interrupt apprises occurrence of most recent sampling moment of the FPGA. The one-off interrupt is intended to bring a periodic control task into correct phase with the FPGA.

3.4.5 User Interface

The user interface is a command based messaging system, where the OSFW requires that a command is passed from the FDSP via the SPI as a string. Current version of the software doesn't support datagram packaging as no interface protocol is required nor used. Thus, additional development is left open to be conducted in future projects. The principle of the command API's operation is to utilise a separate, user defined command structure, which is linked to a task via the task's pointer argument. This way commands can be directed straight into the task, thus additional coding is required to define how a command is used in a task.

The command API is designed to accept a command, check the command's validity, process it by writing parameter values, and release temporary memory reserved for a command object by the communication API. Commonly, the communication API parses and packages a command string into a command object, reserves temporary memory from a command buffer, and signals the system's command handler task. All internal signaling between the communication and command APIs is carried out via the µC/OS-II queue services. In Figure (3.6) is presented a command and task's data objects.

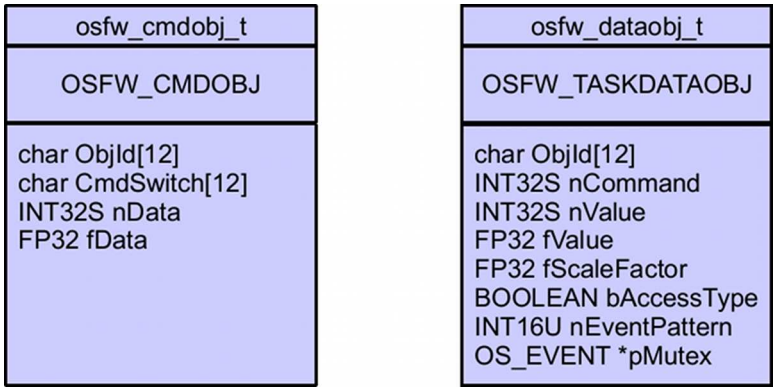

Figure 3.6: Diagrams of the OSFW's command buffer object and unified task data object.

Task's data object is the substantive command interface as the command handler task interprets the command object's string command and writes an equivalent integer command into the data object along optional general use parameters (nData, fData) or alternatively executes an data object related command, for instance sets a scale factor. The data object can be locked upon an object creation, thus making it read-only. If the object needs to be updated and used as parameteric command object, it's declared writable. Available commands of the current version of the OSFW are presented in Table 3.6.

Table 3.6: User interface commands of the OSFW 0.9.0

Command	Explanation
set	Sets an integer value multiplied by scale factor
setInt	Sets a plain integer value
setReal	Sets a plain real value
setScl	Sets a scale factor
setEflag	Sets an eventflag bitmask
run	Sets nCommand -> RUN
runTime	Sets nCommand -> RUNTIME and sets nValue as preset time (µSec)
runWindow	Sets nCommand -> RUNBUF and sets nValue as windowed buffer width
print	Sets nCommand -> PRINT. (To be used as print command in task code)
stop	Sets nCommand -> STOP
resetVal	Resets parameter's current values (integer, real)
printVar	Print parameter's (A parameter id without switches does the same)

3.4.5.1 Implementation

The command API hides implementation details, thus using the interface is straightforward as an object declaration and additional control object implementation coding is all that is required to fully utilise the command API. The command object can be used in user written task or function providing that address of a data object is passed into a utilising function. Figure 3.7 shows the command interfacing semantics.

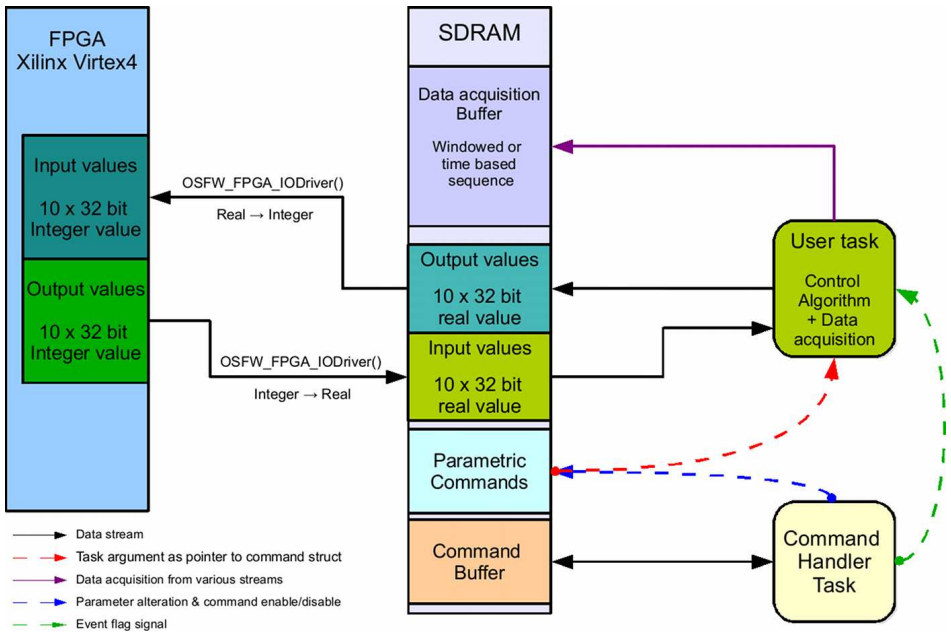

Figure 3.7: A schematic diagram of the OSFW's command interface about command introduction into a periodic, data collecting control algorithm.

3.4.6 µC/OS-II Extensions

Programmable extension hook functions for µC/OS-II are an adjunct feature to the normal RTOS functioning as the architecture of µC/OS-II promotes a possibility to call additional low level functions upon specific RTOS operations, for instance a context switch. Therefore, global interrupt enabling and disabling functions of the RTOS are enhanced with additional function calls; `OSEnterCriticalHook()` and `OSExitCriticalHook()` respectively. These hook functions affects on to EMIF controller's NMI interrupt generation. Thus, the hooks are meant for advanced interrupt management as the EDC-DSP board's signalling connection between the FPGA and DSP requires additional control capability. Assembly code for interrupt handling and code of the OSFW's extension hooks is presented in Appendix IV, 1.

3.4.7 Periodic Task Execution

Periodic code execution is the key factor in the OSFW software development as one of the initial requirements is to come up with a solution that can provide a robust and deterministic on-time execution for a periodic task. Generally, commercial grade real-time operating systems offer some API functions to generate delays in task execution. Typically, a RTOS delay function is based on counting a tick value, where a counter value passed by the delay function is decremented as part of the system's tick service function. This approach is not desired in hard real-time systems, where a task needs to be executed after a deterministic fashion at fixed intervals. The tick based delay functions cannot launch exact imperative task switch in favour of a delayed task as the delay resolution is not generally accurate. For example, fidelity of periodic delay sways between 0 and one system tick in µC/OS-II [1, p. 145–151]. Therefore, even a high priority task's execution cycle with a common, time based delay function is prone to fluctuation resulting from dispatcher's scheduling decisions and occurrence of served interrupts.

Additionally, time based delay functions may create additional jitter caused by timer value calculation in high resolution systems,wherein tick frequency is in kHz domain. A deterministic, fixed rate periodic execution can be achieved by using hardware counter interrupts if used RTOS environment supports low abstraction level programming, for instance hook functions. In connection with Micrium's µC/OS-II real-time operating system selection as basis for the OSFW software development, all necessary software components to program a periodic extension for the OSFW are provided.

3.4.7.1 The Periodic Extension for µC/OS-II

A need for a task execution at deterministic frequency in embedded software development is incessant, thus a solution for portable and robust periodic code execution for µC/OS-II has been proposed in [4]. The periodic extension is designed to utilise a task's control block (TCB), where a defined data structure holds values for task's cycle count and current value of a periodic counter. Upon hardware timer's interrupt, the system's tick service routine calls extension's timer tick hook function, which updates counter values in the data structure and returns a periodic task into ready-to-run state, if the counter value turns zero.

Although the periodic extension is robust, portable, and leaves the source code of the RTOS intact, a complication appears in a form of erroneous period with the first cycle after a task suspension. This is due internal pursuit of µC/OS-II operation semantics. The periodic extension is included into the OSFW code base as a functional expansion and all of the OSFW's cyclic system tasks make use of it. Implementation of the periodic extension requires that the RTOS configuration parameters are set as [4, p. 4–5]:

```
OS_CPU_HOOKS              0
OS_TIME_TICK_HOOK_EN      1
OS_TASK_CREATE_EXT_EN     1
```

Where the CPU hooks selection parameter states that additional hook functions are not placed in the processor port source code files but declared separately in the OSFW's main header file. The tick hook parameter is an adjunct configuration parameter added for the periodic extension and needed for correct operation of the system. The complement task creation parameter is required for proper task creation as a task control block must be cleared upon a task generation [4, p. 1–4]. The periodic extension is composed of two functions, which effectively forms the extension's interface; `OSTaskMakePeriodic()` and `OSTaskWaitPeriodic()`.

A task is transformed into a periodic executable by calling the `OSTaskMakePeriodic()` function, which prepares the task to use the periodic extension. This function must be called prior the task's interminable loop. The task is made to execute in cyclic manner by calling the `OSTaskWaitPeriodic()` function in periodic code section, that is the loop. For example, a watchdog task of the OSFW uses the periodic extension with a smallest possible tick count. The program code of the watchdog is presented on Appendix V, 1.

3.5 Application Program Interfaces

The application program interface of the OSFW is divided into five divergent API sections; command, error, communication, memory, and buffer. The classification is based on operational characteristics of the divergent APIs. Thus, labels of the APIs depicts more of semantic activity than strict sealed program entities. Assignments of the OSFW's application programming interfaces are presented in Table 3.7.

Table 3.7: Priorities of the OSFW's application programming interfaces.

System API	Definition of API and typical services
Communication	Manages all communication traffic in and out of the system, passes processed command objects into the command API. Provides low level driver for the FPGA $\leftrightarrow$ DSP data transfer and a function to send data out of the OSFW.
Command	Processes command objects sent by the communication API and manipulates task related commands in accordance to issued commands. Provides tools for a command creation.
Error	Looks after for internal software errors recovery and handles communication device errors, for instance SPI. Provides centralised error handling interface and management for user applications.
Memory	Provides wide array of memory accessing macros and functions for register access, bit manipulation, and memory block management. Block writes are performed in secure manner as the block writing functions checks the memory map's restrictions
Buffer	Provides a versatile buffering interface for various data collection needs and tools to create, manage, and delete a buffer.

Utilising an application programming interface of the OSFW is straightforward as all functions are documented thoroughly and the APIs are mainly utilised via few interface functions. Generally, services provided by µC/OS-II, for instance queues and mailboxes are used in hidden manner within the system APIs. This architectural design promotes software modularity and portability as the OSFW can be conveniently affiliate with a novel RTOS if needed. The design ensures that all internal data exchange is carried out in secure manner as every data item is placed in individual dynamic memory block and pointer of the address is transmitted via the RTOS queues. Therefore, the system tasks can remain idle until an assigned data object that needs to be served invokes the task, thus saving clock cycles for user application code. The OSFW's API structures are presented briefly in the following sections. However, a complete reference guide, written by the author, is available [22].

3.5.1 Memory API

The Memory API is a collection of functions and macros that forms a memory abstraction layer for the framework. The main purpose of the API is to provide higher abstraction level memory accessing tools to safely conduct memory operations in 8 and 32 bit registers and as well with larger, user defined memory blocks. The API offers also a pack of conversion functions to perform variable data type transformations. The API layer is compounded of four semantically different memory operations and programmed to be as general and unconnected to the underneath µC/OS-II operating system as possible. The usage of the memory API reverts to utilisation of an individual function, which underlines the API's role as an abstraction layer for memory and register operations.

3.5.1.1 Controlled Memory Access

The OSFW's memory accessing functions are based on the standard C (C89) functions. However, the functions utilise control routine to verify whether a memory address is valid in relation to defined address space limits. Hence, a target address is checked, whether it contains enough free space in accordance to RAM, SDRAM and the FPGA memory limits. If size of data is too large to fit into a destination address, a memory access error is returned [22, p. 67–70]. The low level memory functions of the OSFW are:

- `INT8U OSFW_MemCheck (void *pAddr, INT32U *pSize)`
- `void* OSFW_MemClear (void *pAddr, INT32U *pSize)`
- `void* OSFW_MemCopy (void *pDest, void *pSrc, INT32U *pSize)`

3.5.1.2 Macros

The memory API provides two general use utility macros; `OSFW_SetBit(addr,bit)` and `OSFW_ResetBit(addr,bit)`) for memory address bit level manipulation to be used with user defined variables. Owing to the structures of DSP's registers, these macros may not be used to access hardware control registers of the DSP.

3.5.1.3 Bit Manipulation

The register manipulating functions provide a bit level access to memory mapped registers of the DSP. These functions can address any DSP's memory mappable register in accordance to the register access characteristic [8, p. 15]. However, the validity of an address argument is not checked, as it's assumed that these functions are used mainly in the system's initialisation phase. The bit manipulating functions are [22, p. 81–85]:

- `BOOLEAN OSFW_TestBit (volatile INT32U *pAddr, INT8U bit)`
- ```
  void OSFW_SetWordRegisterBit (volatile INT32U *pAddr,
                                INT8U bit, INT8U mode)
  ```
- ```
  void OSFW_RegisterBit (volatile INT32U *pAddr,
                         INT8U bit, INT8U mode)
  ```
- ```
  void OSFW_ResetWordRegisterBit (volatile INT32U *pAddr,
                                  INT8U bit, INT8U mode)
  ```
- ```
  void OSFW_SetByteRegisterBit (volatile INT32U *pAddr,
                                INT8U bit, INT8U mode,
                                INT8U RegOffset)
  ```
- ```
  void OSFW_ResetByteRegisterBit (volatile INT32U *pAddr,
                                  INT8U bit, INT8U mode,
                                  INT8U RegOffset)
  ```
- ```
  void OSFW_SetWordRegisterMask (volatile INT32U *pAddr,
                                 INT32U Mask)
  ```
- ```
  void OSFW_SetWordRegisterMask (volatile INT32U *pAddr,
                                 INT32U Mask)
  ```
- `void OSFW_SetRegister (volatile INT32U *pAddr,INT32U Val)`
- `void OSFW_ClearInterrupt (INT8U Interrupt)`
- `void OSFW_EnableInterrupt (INT8U Interrupt)`
- `void OSFW_DisableInterrupt (INT8U Interrupt)`
  ```

3.5.1.4 Data Types Conversion

Conversion functions provide a way to convert variables from one data format to another. Although not all functions are currently needed, a plethora of conversion algorithms are supplied for future needs. The conversion functions are selected to cover the most typical data type conversion requirements [22, p. 56–66]:

- `FP32 OSFW_IntToFloat (INT32S *pValue, FP32 *pSclFactor)`
- `FP32 OSFW_LongToFloat (long int *pValue, FP32 *pSclFactor)`
- `FP64 OSFW_IntToDouble (INT32S *pValue, FP32 *pSclFactor)`
- `FP64 OSFW_LongToDouble (long int *pValue, FP32 *pSclFactor)`
- `INT32S OSFW_FloatToInt (FP32 *pValue, FP32 *pSclFactor)`
- `INT32S OSFW_DoubleToInt (FP64 *pValue, FP32 *pSclFactor)`
- `long int OSFW_FloatToLong (FP32 *pValue, FP32 *pSclFactor)`
- `long int OSFW_DoubleToLong (FP64 *pValue, FP32 *pSclFactor)`

3.5.1.5 Miscellaneous Functions

The miscellaneous memory API functions offers general tools to perform approximation delays in a user code and a method to access the DSP's internal watchdog timer. Loop delays that doesn't utilise µC/OS-II delay, are typically used in initialisation phase. Thus, a 'no-operation' loop function is provided for such needs. The digital watchdog's hardware counter can be programmed with a provided interface function and the same tool can also be used to initiate software based hardware reboots. The miscellaneous memory API functions are composed of [22, p. 71,119]:

- `void OSFW_NopDelay (INT32U Delay)`
- `void OSFW_C6727_Watchdog (INT8U mode)`

3.5.2 Error handling API

The error handling API is basicly an interface that transfers an error object from a task to the error handler task. The interface is programmed to provide centralised error printing capability within the OSFW. Additionally, the error handler task can be used to reset the system. A compiler directive is provided to control whether to print an error message or not. From a developer point of view, a declared and set up error data object is all what is needed to utilise the API. Once the error data object is initialised, additional call of `OSFW_ErrorReport()` function with an associated error object is needed to generate an valid error message which is handled by the API. The OSFW's error API utilisation is presented in Appendix IX, 1. The service layer is composed of following functions and tasks [22, p. 8,58,59,156–159]:

- `void OSFW_ErrorQueueInit (void)`
- `void OSFW_ErrorMsgGen (OSFW_ERRDATA *pErrData)`
- `INT8U OSFW_ErrorReport (OSFW_ERRDATA *pErrData)`
- `void OSFW_TaskErrHandler (void *pdata)`
- `void OSFW_TaskSPIErrHandler (void *pdata)`
- `void OSFW_TaskWatchdog (void *pdata)`

3.5.2.1 Error Object

The error object is a composite structure, which holds data items for; function identification, error code, and error class. These parameters defines error event's source function, severity, and classification. The error object is presented in Figure 3.8.

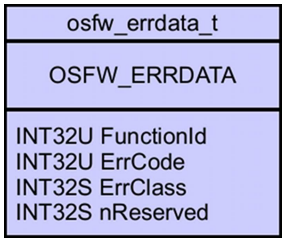

Figure 3.8: A detailed diagram of the error API's data object.

3.5.3 Communication API

The communication API is a service layer composed of hardware and class drivers, data parser, checksum routines, and data transfer tasks. The API is dependent on many services provided by µC/OS-II, thus as part of the API, several initialisation routines are provided to set up the required services. The main purpose of the API is to provide effectual means to handle data flow between the framework and the hardware. The architecture of the API is designed and programmed to be flexible, coherent and object based. All data exchange is carried out in dynamically allocated memory, thus every communication object resides in its own memory block. The required memory is allocated either from Tx or Rx buffers, thus mutual exclusion for data is effectively ensured. Hence, encapsulated datagrams are essential part of a pointer based data transfer layer throughout the OSFW and µC/OS-II.

Another reason for the selected convention is to make it possible to enhance the communication objects to carry divergent data, as the API interface remains the same. The most of the API is hidden leaving only the `OSFW_CommDataSend()` and `OSFW_printf()` functions at developer's disposal. Generally, only the printing function is needed as it provides a user friendly way to pass a string message through the transmission path. Currently, all inputs are interpreted as commands, thus in practice Rx data is handled by the command API. The communication API is composed of following functions and tasks [22, p. 50–53,60,80,86,154,155]:

- `void OSFW_CommBufQueueInit (void)`
- `INT8U OSFW_EventFlagInit (void)`
- `INT8U OSFW_CommDataSend (OSFW_TXOBJ *pDataTx)`
- `INT8U OSFW_CommDataParser (OSFW_RXOBJ *pDataRx)`
- `INT8U OSFW_CRC (char *pStr)`
- `void OSFW_RxHwDriver (INT8U RxBus)`
- `void OSFW_TxHwDriver (INT8U TxBus, void *pDataTx)`
- `void OSFW_TaskCommTX (void *pdata)`
- `void OSFW_TaskCommRX (void *pdata)`

3.5.3.1 Communication Drivers and DMA

The hardware communication devices are programmed jointly with the driver layer to employ direct memory access services to shift data fluently and to avoid extensive interrupt burden on CPU. Commonly, the drivers interact with the hardware transmission settings and the communication buffers, while DMA transfer takes care of actual data shifting without CPU interference. Still, few interrupts are needed serve the hardware communication peripherals: SPI, I^2C, and dMAX to complete Tx and Rx DMA messaging services [27, 28].

3.5.3.2 Data Transmission

Data transmission between the OSFW and the FDSP is done by dMAX via the SPI 1 bus, where the class driver task; handles the transmission queue, calls the hardware driver, waits until a data has been sent and releases a memory block of an allocated Tx object. The SPI 1 is currently designed to operate as master output bus. Therefore, the hardware driver sets up dMAX transfer entry table for the SPI slave transfer, which can be used also with the master protocol. Upon an operation cycle; the driver disables the SPI 1 bus event generation, sets up the data source and count values, enables bus events and writes a kickstart byte to the SPI 1 shift register to initiate DMA transfer. However, the SPI 1 or any other dMAX transfer completion event initiates a collective interrupt, thus CPU event status register is checked in accordance to DMA event entry table to clear any associated interrupts.

3.5.3.3 Data Reception

Data reception is currently configured to use the I^2C bus as an example for further development resulting from incomplete communication algorithms in the FDSP's software. The I^2C hardware is configured to generate an interrupt for every received character, which is stored into a static reception buffer. When a string termination marker byte is received, the content of the buffer is copied and transferred to the Rx class driver for further processing. This method is inefficient and causes unnecessary load on CPU. Concerning the OSFW, the I^2C bus will be decommissioned in future as the SPI 0 bus reception algorithm and its DMA employment are in high priority in further development intents.

3.5.3.4 Data Parser

All inputs passed into the OSFW are currently handled as commands only, thus received data has to be parsed to get a command identification string, command switches and parameters to formed up as a command object. The `OSFW_TaskCommRX()` system task is a class driver, dedicated to manage reception data. The task calls the parser function, which in turn parses an input message string, constructs a command object and puts it into the command queue. Commonly, it's assumed that the parser's argument is a plain ASCII string without any datagram headers. The valid format of the command string is:

$$CommandId\ \text{-}CmdSwitch(n)\ Operand(n) \ldots \text{-}CmdSwitch(n{+}11)\ Operand(n{+}11)$$

Where $CommandId$ is an identification string of the OSFW residing command object (Figure 3.6) and $CmdSwitch$ is a hardcoded command with associated parametric value $Operand$. Even though the term 'command' is applicable concerning the task's data object, it actually means the identification string field of the task control object. Therefore, the command switch field holds the actual operative command. In addition, up to a twelve commands can be chained into a single command string as this feature is provided for situations, when several operations within a task needs to be carried out at short notice. Basicly, the data parser is a connection point between the communication and command APIs, as the command API gets all command inputs via this route.

3.5.3.5 FPGA Data Driver

The FPGA data driver is a part of the communication API providing an atomic data exchange and data type conversion between the FPGA and DSP. The design of the driver allows variety of applications to utilise the driver in data connections with the FPGA. The driver can be used as stand-alone data converter without initialisation to exchange data between virtually any mappable memory. The FPGA Driver is composed of:

- `INT8U OSFW_FPGA_IOMemInit (void)`
- ```
  void OSFW_FPGA_IODriver (OSFW_DSP_IO_DATA *pDSP,
                           OSFW_FPGA_IO_DATA *pFPGA,
                           OSFW_FPGA_IO_CNTRL *pCntrlObj,
                           BOOLEAN mode)
  ```
  ```

3.5.4 Command API

The command API is an object based service layer that makes it possible to create parametric global command objects, which are used to pass commands into a task. Commonly, the interface of the API is a task control object (Figure 3.6), which has to be created and initialised prior employment. Upon an object creation, a dynamic memory block is allocated from the task data area memory pool and the object's address is placed into a free element of the data-object pointer array. The task control object can be initialised as either read or read-write type. Hence, if the choice is read-write, a mutex semaphore is acquired during the object creation and the address of a mutex is written into the object's mutex pointer element. Otherwise the pointer element's address is nullified.

Task control objects that are accessible by a user are inheritly protected from the shared data problem, as long as mutexes are acquired while accessing the task's data objects. The object is designed to be suitable for the OSFW's current and future command processing needs. Additionally, developmentability is ensured by reserving additional space on the OSFW's memory map. The command API is composed of following functions and a task [22, p. 48,49,54,153]:

- `void OSFW_CommandQueueInit (void)`
- ```
  void OSFW_DataObjCreate (char *pCmdID, INT32U nCommand,
                           INT32S nVal, FP32 fVal, FP32 fScl,
                           BOOLEAN bType, INT16U nEvent,
                           INT8U TaskPrio, INT8U *Error)
  ```
- ```
  void OSFW_CommandInterpreter (OSFW_CMDOBJ *pCmdObj,
                                OSFW_TASKDATAOBJ *pContrObj,
                                INT32U nSwitchIndex)
  ```
- `void OSFW_TaskCommandHandler (void *pdata)`

Using the API requires a task data object declaration and creation. If the object is to be used as a command interface within the task, the return value of the command data object creation function `OSFW_DataObjCreate` must be used as `pdata` argument of the `OSTaskCreate()` function. After the setup, the task control object can be addressed and used with the command API services [22, p. 16].

3.5.5 Buffer API

The buffer API is a data-buffer system intended be used with various data acquisition routines. The API is an object based ring-buffer architecture equipped with general tools to manage buffer allocation and deletion. The buffers of the API are designed to be dynamic and versatile to employ, but can be utilised partially without dynamic allocation in static manner. A ring-buffer object is a declareable variable that is composed of; buffer pointers (head,tail), size variable, and from the buffer itself, which is an array of void pointers. The API provides an all-around buffer utilisation, meaning that only buffer pointers are adjusted and if a buffer becomes full, the oldest sample is replaced with the newest input data. Purpose of the API is to provide data buffers for windowed or patch driven data recording. The ring-buffer object is initialised into a static size, which dictates its maximum usable size. The size can be anything between the predefined maximum or minimum values. This makes it possible to change a buffer's size, that is usable array area, on the fly. Any type of data including variables, structures, or strings can be put into a buffer in accordance to the buffer object's size limits. The ring-buffer API is composed of following functions [22, p. 75–79]:

- `INT8U OSFW_RingBufferInit (OSFW_RINGBUFOBJ *pBuf, INT32U BufSize)`
- `INT8U OSFW_RingBufferCheck (OSFW_RINGBUFOBJ *pBuf)`
- `void OSFW_RingBufferAdd (OSFW_RINGBUFOBJ *pBuf, void *pData, INT8U *pError)`
- `void OSFW_RingBufferClear (OSFW_RINGBUFOBJ *pBuf, INT8U *pError)`
- `void* OSFW_RingBufferCreate (INT32U BufSize, INT8U *Err)`
- `INT8U OSFW_RingBufferDelete (OSFW_RINGBUFOBJ *pBuf)`

Using the API is very straightforward and broadly quite similar to µC/OS-II services in general. As the interface is somewhat object-oriented, it's mandatory to declare and initialise the buffer object differently for static and dynamic use. A dynamic buffer object resides in SDRAM, while a static object is declared as a local or a global variable. Commonly, usage of the buffer API is down to a one function: `OSFW_RingBufferAdd()`, which checks the buffer's state, allocates memory for data, if needed, and rolls tail and head values of the buffer.

4 Development of the OSFW

The OSFW is programmed in various phases starting from preliminary study concerning the system's requirements and ending up to platform specific port testing. The principal work is done during flow diagram design, whereby the most fundamental software structures are resolved. Commonly, programming of the OSFW is performed in two phases, in which primary programming and development was carried out with GNU Compiler Collection (GCC) IDE in Linux environment and additional programming took place while fitting the software into the target DSP in the CCS environment. Therefore, programming and documentation of the OSFW is divided into several milestone steps, in which informal version management methods are emphasised as well as arrangement and contents of source code files. Generally, The OSFW's software development is compound of making of a documentation and writing program code simultaneously as one of the key principles set a condition that documentation must provide wall-to-wall information about the OSFW to ensure developmentability and ease of maintenance now and in future. Therefore, the programming style used with the OSFW ensures that documentation incorporates every aspect of the program code. All documentation is written directly into the source code including Doxygen format documentation [22] and general developer notes, which in turn emphasise specific details of the code.

Evolution of the OSFW took place in two distinct IDEs; GCC and CCS, which is not an issue as embedded software is rarely developed in a target device. The GCC and Linux environment offers an expansive set of programming tools at developer's disposal, hence the most of the OSFW's code is programmed and tested in Linux by utilising various software development tools such as an open source debugging utility Data Display Debugger (DDD). As the OSFW is strictly C89 compliant, the C compiler of the GCC made it possible to compile the software in accordance to the standard C in pedantic manner. Therefore, the program code parts, which were developed in Linux, fit in the CCS environment comfortably, where final assembly and testing of the OSFW is made.

4.1 Perspective of Documentation

The premise of OSFW software documentation is that it must be extensive, clear and effectually generated. Therefore, all documentation is embedded into the source code in one of Doxygen's supported documentation format; JavaDoc. An external documentation tool makes the documentation more outlined as all efforts can directed to contents generation instead of fashioning document's exterior features. Automated document generation is an asset as a unique document can be formulated for every new software revision. Hence, virtually everything is documented in the OSFW source codes as all globally declared definitions, variables, structures, enumerations, and functions are commented thoroughly in accordance to the documentation requirements. However, local variables of the OSFW's functions are not included in the documentation as their usage is confined to a single function. Therefore, additional ordinary commentary is added to enhance understanding and to point out specific details on the source code. Thus, documentation of the OSFW is provided as two medium; web page in Hyper Text Markup Language (HTML) and 192 pages long portable document file (PDF) reference manual [22].

4.1.1 Doxygen: The Source Code Documentation Generator Tool

Doxygen is a multi-platform open source documentation tool for C, C++, Java, Objective-C, IDL, Fortran, VHDL, and C# programming languages. Commonly, Doxygen supports various output formats from on-line documentation in HTML to off-line reference manual in LaTeX. In addition, the program supports various, broadly used publication formats such as; Microsoft's rich text format (RTF), PostScript, hyperlinked PDF, and UNIX® manual pages. One of Doxygen's strong areas is that LaTeX formulation and commands can be included in the commentary [18]. Therefore, Doxygen provides effectual environment to produce disparate documentations. In addition, graphical tools are provided to forge source code visualisation for structures, classes, and function call and caller relations diagrams. Typically, Doxygen output is generated according to a control file (Doxyfile), which incorporates options to manage output's appearance. To utilise Doxygen, source code's commentary must be formed after a distinctive fashion to distinguish between ordinary commentary and intended documentation. One of available commentary styles (JavaDoc) is presented on Appendix V, 1. In addition, Doxygen's output of the watch dog task's commentary is presented on Appendix VII, 1.

4.2 Linux Programming Environment

GNU/Linux environment got selected as principal development platform owing to nature of the OSFW for being a hardware independent service layer for µC/OS-II as the framework doesn't require hardware specific IDE for software development. Additionally, previous knowledge and experiences gained from µC/OS assured that the OSFW can be developed without close interaction with the RTOS in initial development cycle. Although µC/OS-II is highly portable, there was no working port available at time for x86-64 architecture that could operate on Core™ 2 Duo processor by Intel® in 64-bit environment as the development platform PC is of the type.

Owing to characteristics of the Code Composer Studio, the preliminary study settled that C89 is the standard to be used in development of the OSFW. Therefore, the GCC's C compiler was utilised extensively to develop C89 compliant program code as the compiler was set to warn about non-starndard programming. Generally, the GCC is high quality utility, which conforms to programming standards closely. The GCC was used to produce x86-64 executable and linking format (ELF) binaries, which were tested in Linux environment. Concerning the OSFW, this is not an issue as x86-64 code is needed only to test software's semantics. However, clearly malfunctioning code would have biased the course of programming, thus a mature stable version of the GCC was utilised for the purpose. Hence, all the OSFW specific structures, algorithms and general functionality are compiled and tested in Linux, regardless of the code being a 64-bit x86-64 machine code. In Table 4.1 is presented tools used in development and programming of the OSFW.

Table 4.1: Programs of Linux environment used in the OSFW development and programming.

Program	Version	Usage definition
GCC	4.1.2	GNU Compiler Collection is used to check code syntax and compile test versions of the OSFW's functions.
GDB	6.8	The GNU debugger is used to debug and check a test function's internal functioning.
Valgrind	3.4.0	Valgrind is used to reveal memory leaks and uninitialised memory sections during run-time.
DDD	3.3.11	Dynamic data debugger is a GUI for the GDB and used to visualise program code operation
Kate	2.5.10	Kate is a versatile text editor geared towards software development. The OSFW is initially written with Kate.

4.2.1 Algorithm Development and Testing

Virtually all algorithms and functions of the OSFW are developed with the GCC by utilising disparate compiling options. Generally, algorithms are first planned and then programmed, compiled and tested, re-programmed if needed, and conjoined as a function. This iterative and incremental development methodology proved to be efficient as all algorithms of the OSFW were tested early on, thus ensuring that the most fundamental operational parts of the code are substantiated correct with debugging tools, that is Valgrind and DDD. Upon software testing, the OSFW's sources codes were compiled with command options:

$$gcc\ \text{-}g\ \text{-}Wall\ \text{-}Wextra\ \text{-}pedantic\ \text{-}std\text{=}c89\ <program>.c\ \text{-}o\ <program>$$

Where $Wall$ enables the most available warning options and $Wextra$ provides additional warning messages for various events, for instance a comparison between signed and unsigned values. The $pedantic$ option is used to engage all mandatory diagnostics recited in the C standard, However, the option doesn't necessarily produce outright ISO C compliance in program code as some non-ISO conventions are found but not all [29, p. 49]. The std command switch is used to inform the compiler that the source code is supposed to conform to a set standard. The o parameter denotes output file, where a compiled program is saved. The g option includes debugging information in the system's native formats, for instance common object file format (COFF) or debugging with attribute record format (DWARF). Aforementioned debugging formats are supported by the GDB. Additionally, the compiler option actually produces extra information that only the GDB can utilise [29, p. 32,50–57,65].

After a successful compilation, that is no errors with selected options, the code's functionality is tested with Valgrind memory debugger to ascertain that program doesn't produce memory related errors, for instance memory leaks. Thus, Valgrind's memory checking features proved to be crucial as nearly all obscure program execution errors were found with the tool. Commonly, an algorithm functions well, if it's performance is consistent and triggers no errors in Valgrind's tests. Typically, majority of memory allocation errors reported by Valgrind are related to uninitialised pointers, local variables and failed memory release. However, depending on GNU/Linux system's resources, for instance glibc, Valgrind may report errors that occur within the system's library functions. Therefore, caution should be exercised while interpreting Valrind's output. Commonly, this isn't a problem in GNU/Linux systems, which utilise the glibc library supported by Valgrind.

Dynamic Data Debugger is used to visualise program's inner operation, providing complete insight of functioning of an algorithm. DDD makes it possible to view source code, machine code, and program's variables in very detailed manner. Concerning the OSFW, DDD proved to be the most effectual programming tool as un-expected program behavior could be studied very delicately. Hence, the OSFW's software semantics starting from a single algorithm to API layers were surveyed with DDD.

4.2.2 Valgrind - Memory Debugging Tool

Valgrind is a compilation of simulation-based debugging tools, which are used to check program code's performance concerning; memory usage, cache optimization, caller – callee relations, heap usage, and synchronisation errors. Valgrind is strictly Linux software development tool as it's bound to supported CPU architectures, for instance x86-64, and Linux operating system's compiler and the C libraries [30, p. 15]. Therefore, results gained with Valgrind may not apply to embedded software, which is meant to be used in substantially divergent CPU architecture.

Upon testing Valgrind takes control of a test program and simulates it on a synthetic CPU, thus running the code with selected debugging tool. Every instruction of a program is simulated, thus any dynamically linkable library is included in a test [30, p. 16]. By default, Valgrind uses the memory checker tool to provide output of memory leaks and uninitialised values. Typically, Valgrind is invoked with a command:

$$valgrind \;\; \text{--}leak\text{-}check{=}yes \;\; \text{--}track\text{-}origins{=}yes \;\; < program >$$

Where $leak\text{-}check$ option executes memory leak search when a client program finishes. The switch has options; no, summary, yes, full, in which 'yes' yields detailed error information on each found memory leak. The $track\text{-}origins$ option turns on tracking for source of uninitialised values, thus providing detailed insight, where an error occurs [30, p. 44]. In Appendix VIII, 1 is presented a typical output of Valgrind's memory checker tool.

4.2.3 Data Display Debugger

DDD is a graphical front-end for a selection for a variety of command-line debuggers; GDB, Debugger for Unix (DBX), and Ladebug, of which the GDB is recommended to be used with the program. Generally, the program supports variety of configuration options ranging from X display option to controlling selected internal debugger, thus preparing the way for individualised debugging environment. In addition, DDD can be configured to use a remote internal debugger, for instance having a centralised debugging server for a group of developers. User interface of the program is intuitive and it provides clear layout of a program's source code, machine code, pointer linkage, local and global variables, and argumental data. The program execution window (terminal) along the GDB console can be utilised during software tests, if needed. In order to debug a program, it must be compiled with debugging information, for instance the GCC's g option, thus only correctly compiled programs can be tested with required debugging information. Using the program is straightforward and efficient as the GUI provides visual cues for program steps, watch and breakpoints, and variables' current values, thus making a program's execution tracing convenient. In addition, debug sessions can be saved and loaded, which gives a possibility to compare different versions of a same program in DDD in case of persistent software error [31]. In Figure 4.1 is presented a graphical user interface of the program.

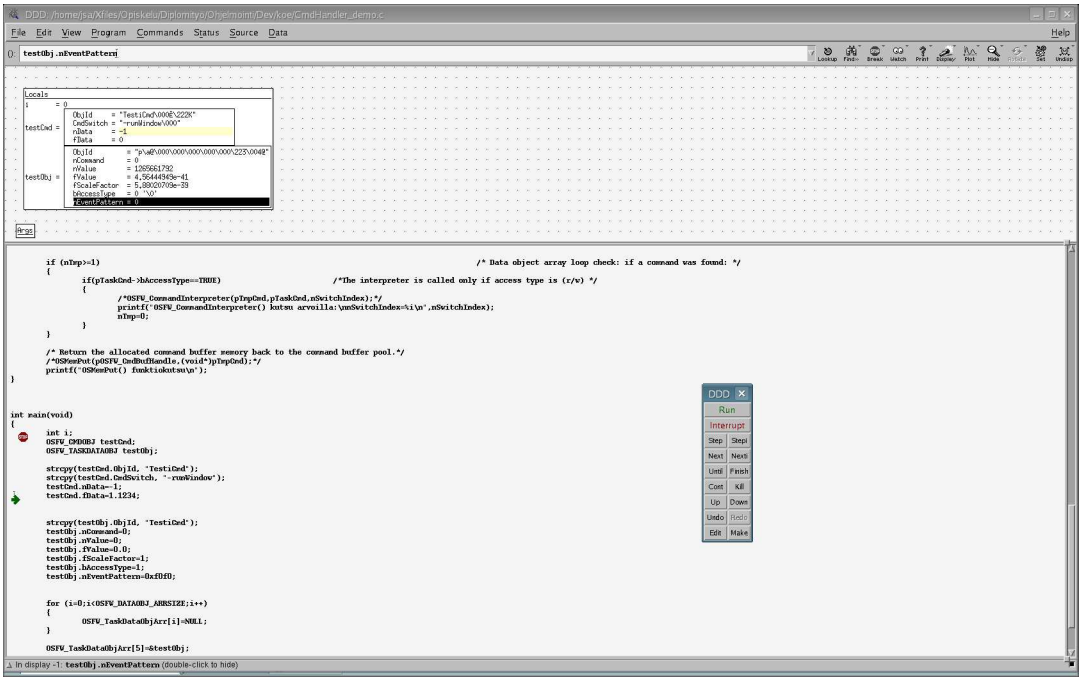

Figure 4.1: Screen layout of Dynamic Display Debugger showing a test of the OSFW's command handler task, wherein source code along local variables and arguments are visible.

4.3 Assemblage of the OSFW

The Code Composer Studio became as principal development environment after the comprehensive programming done in Linux environment because conjoinment of µC/OS-II and the OSFW couldn't be efficiently set off in Linux owing to an incompatible RTOS port. Therefore, the CCS is mainly utilised to forge and test integration of the OSFW and µC/OS-II. The compiler of CCS was used after a similar fashion as the GCC; to root out syntax errors in the OSFW's source code. Hence, an early goal was to get the OSFW to compile correctly in the CCS, followed by actual algorithm and function testing. Development efforts done in Linux proved to be significant as the RTOS and OSFW integration and most of the DSP specific initialisation routines compiled with little or no errors. However, some algorithms were re-programmed as the CCS's C89 compliant compiler didn't accept either syntax or some other more or less uncharted factors of the code, which functioned fine with the GCC. Commonly, the OSFW is assembled in the CCS as all functional parts are tested and tuned separately and in gear, starting from low level initialisation routines and ending up to the API semantics. Essential compiling options used in the OSFW development are listed in Table 4.2.

Table 4.2: Consequential configuration options of the CCS' compiler used in the debugging phase.

CCS Compiler Configuration		
Basic Options	Target Version	C67x+
	Debug Info	Full symbolic
	Optimize for Size	No
	Optimize for Speed	5
	Opt Level	Function (-o2)
	Program Level Opt	None
Advanced Options	RTS Modification	No RTS Funcs
	Auto Inline Treshold	None
	Endianness	Little Endian
	Memory Model	Far Aggregate
	RTS Calls	Use Memory Models
	Aliasing	Default
Parser	ANSI Compatibility	Non-Strict ANSI

Generally, the assemblage of the OSFW was convenient as the most difficult inconsistensies found in the code were linked to hardware initialisation routines. In addition, the table shows that source code level optimizations could be used despite the fact that the OSFW is a beta level software at best, thus denoting efficiency of development done in Linux.

4.3.1 Code Composer Studio

The Code Composer Studio is a composite development environment by Texas Instruments designed for TI's digital signal processors application development. The CCS provides software tools for code generation, real-time debugging with a target DSP or with a simulator, and extensive system analysis. Generally, the CCS manages software development as project, wherein all necessary software and management files such as a linker's configuration file, source codes, external libraries, and additional API sources, for instance DSP/BIOS, are maintained in cohesive manner. For example, an external documentation file can be added as part of project as done in the OSFW project.

The project manager supports application development and compiling with preset configuration domains, for instance the default domains are *Debug* and *Release*, thus providing a possibility to utilise divergent compiling optimizations. Commonly, code generation tools of the CCS are the basis of the development environment including; source editors, the C89 conforming compiler (CCS 3.3), assembler, and linker. Software testing is effectual as the CCS offers hardware emulation debug platform tools. However, not all the DSP families or types are supported. The emulation software provides on-chip register management, breakpoints usage, counters for cycle time measurements, and program loading, and starting, stopping and resetting the DSP [32].

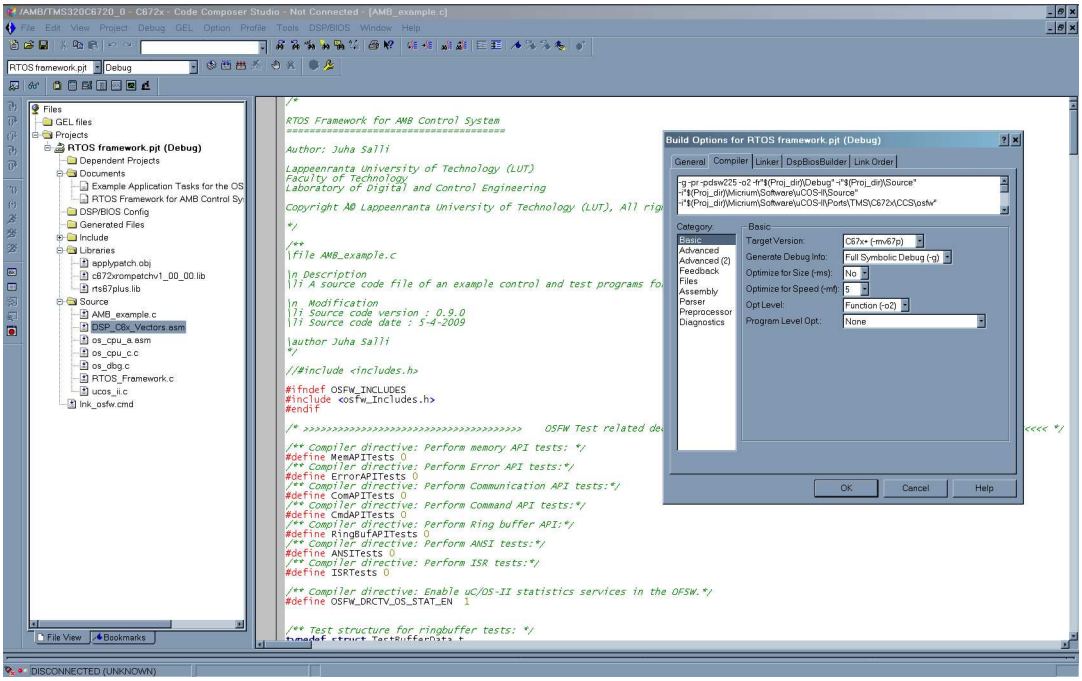

Figure 4.2: Screen layout of Code Composer Studio 3.3 showing an opened source code file and a selection window of the compiler's parameters.

4.4 Testing of the OSFW

Primary testing of the OSFW was carried out in the CCS IDE with the EDC-DSP board, in which the DSP is accessed via the JTAG interface, thus providing efficient and definite platform for testing. Concerning the OSFW's assemblage, all affiliated tests were performed via the JTAG interface as it makes it possible to load the software, administer, and monitor program execution in detailed manner. The JTAG proved to be considerable asset, even though the FDSP provides a binary image loader tool and boot up capability. The SFProg was alpha level software at the time of the OSFW testing, thus making it unsuitable to be utilised as development tool. Hence, all resources were directed strictly to get the OSFW compile and function flawlessly in the DSP of the EDC-DSP board. Testing of actual software semantics began after vigorous programming and compiling phase as no compiling or linking error were accepted in the OSFW source codes or within the linker's configuration. Commonly, assemblage of the OSFW brought additional programming efforts as remaining bugs were fixed and additional coding was done to increase the software's robustness. For example, the most header file inclusion preprocessor macros were added upon initial compiling attempts to make the compiler to avoid multiple inclusion of a same header file.

The software environment of the OSFW is set up by configuring the EDC-DSP board in the CCS's setup configuration tool, wherein a development board with supported DSP family models is defined. Therefore, the configuration isn't an exact duplicate of the existing EDC-DSP board but an semantic layout, where all essential parts are included in accordance to the earlier EDC-DSP board definitions, for instance the *LUT2007* project by LCEDS. After the configuration, the EDC-DSP board is accessible by enabling the JTAG interface of the DSP in the CCS's parallel debug manager.

The EDC-DSP board's JTAG connection requires an additional hardware to provide linkage with the CCS debugging tools. Hence, an USB JTAG emulator adapter; XDS510 USB by Spectrum Digital is connected between PC and the EDC-DSP board to provide the functionality. During the testing, the JTAG interface was prone to malfunction causing a non-operating state, which could be solved only by disconnecting the adapter from the PC and exiting the CCS. This phenomena was rather intermittent as testing could go on for hours without problems but occasionally the adapter could stop responding within few minutes. The issue hindered testing especially in the initial testing phase, when a cause of deadlock was unknown, thus putting the software's general functionality in doubt.

4.4.1 Test Methodology

The OSFW's software is tested in very careful manner as every single source code line's exact functioning is inspected and corrected if needed. Thus, occurrence of register and memory access is checked delicately as the OSFW utilises mostly pointers while addressing the hardware resources. Premise of the method is to ascertain that the software functions as intended, thus resembling practices utilised in algorithm development. Therefore, the testing was straightforward and effortless as most of errors in the OSFW' code were eradicated during initial development in Linux. For example, the system's hardware initialisation routine was executed at a single line at a time and registers and states of affected memory were checked to find indeterminacy in the system's behavior. During the boot up procedure tests it was found that the CCS's compiler didn't initialise an address of interrupt clear register (ICR), thus forbidding the usage of `OSFW_ClearInterrupt()` function, which utilises the ICR register for interrupt clearing. In Appendix X, 1 is presented a test record of the OSFW's initialisation routines.

All functions of the OSFW are tested after an individual fashion by examining a function's operation in detail, while testing the system's application interfaces. The system APIs are tested with a specific application, in which a single API test can be selected at the time with preprocessor macros, thus attenuating unwarranted code upon debugging. The application interfaces of the OSFW utilise functions of each other, thus a test feature of single API is not delimited into a particular API. However, the API functionality depends on correct operation of functions, thus following program flow through the system functions, the correct functionality of the API is ascertained. In Appendix XI, 1 is presented a test record of the OSFW's system functions.

4.4.2 Bootloader Implementation

The bootloader requires that all bootable code must be compiled with `#pragma` preprocessor command, which marks a code section and informs the linker to put it into a preset specific memory section: ".TiBoot". Commonly, only hardware initialisation routines, for instance system clock and memory setup, are included into a boot up procedure [33, p. 15–16]. The FDSP's boot up procedure loads an application program image into the DSP's RAM memory either as ordered by user or upon the board's power up depending on the FDSP's setup. After the application image downloading, the DSP's software bootloader executes boot up code found in the image file, thus starting the OSFW.

The compiler of CCS cannot produce an application image script (AIS) format executable required by the bootloader, hence an external genAIS Perl script is used for the purpose. The application image file is a proprietary binary format file by TI and it's composed of script header and additional bootloader commands, in which each command embodies an op-code with associated data [33, p. 7]. In addition, genAIS script's functioning will most likely require additional Perl package, for example Active Perl, in Windows environment. The OSFW application is transformed into AIS binary package with a command:

$genAis.pl$ -$bootmode\, i2cslave$ -$otype\, bin$ -$cfgtype$ "c" -i <*app.*>$.out$ -o <*app.*>$.bin$

Where $bootmode$ option selects boot mode, $otype$ selects a type of output file, $cfgtype$ determines whether AIS command or application code is used for system initialisation, i and o options declare input and output files respectively [33, p. 32–33].

4.4.2.1 Linker Configuration Issues

During the tests, it was noticed that configuration options of the linker's command file have higher priority than options set in the project's linker configuration window. Therefore, all required patch loading commands, and configuration options are retained in the linker's command file.

4.4.2.2 Phased-Locked Loop Initialisation Issues

The most peculiar bug concerned functioning of phase-locked loop (PLL) initialisation routine, which compiled flawlessly but produced uncertain behavior irregularly, especially during the first boot attempt, but not on the second. Thus, the routine was essentially torn apart and re-programmed in the CCS regardless of initial coding done in Linux. The bug was due to utilisation of external functions within the bootup procedure, because not all required functions were placed into a *".TiBoot"* memory section, thus causing the bootloader to jump program section. Therefore, to avoid unnecessary massive boot loader program section, critical hardware initialisation functions such as `OSFW_C6727_PLL_Init()` and `OSFW_C6727_EMIF_Init(void)` are programmed to access the DSP's hardware resources without utilising any external functions.

4.5 Version Control Issues

Code maintenance issues and version control management are emphasised on the project's requirements because the OSFW's prospective development is planned to come about in decentralized manner. Therefore, the OSFW's project folder's contents are selected after a such fashion that all required software, source codes, documentation, and various software patches are assembled, thus keeping all essential software together. The LCEDS is habituated to utilise an open source code version control software Subversion (SVN) in various projects, thus the OSFW files are formed to fit to be used with the SVN. However, the SVN server of the LCEDS cannot be accessed from external network, that is the internet, because of a security policy. Hence, the SVN was not utilised upon initial software development of the OSFW. The advantages of SVN unfold as the development moves on from testing to production level programming in multi-developer environment.

4.5.1 Arrangement of the Source Codes

The source codes of OSFW are assorted into distinctive parts, that is source code files, to create separation between; base code, hardware related initialisation code, interrupt service routines, µC/OS-II extensions and port files, system tasks, and configuration options. The separation is similar to source code's classification of µC/OS-II. In Figure 4.3 is presented a schematic layout relation of the OSFW and µC/OS-II source code files.

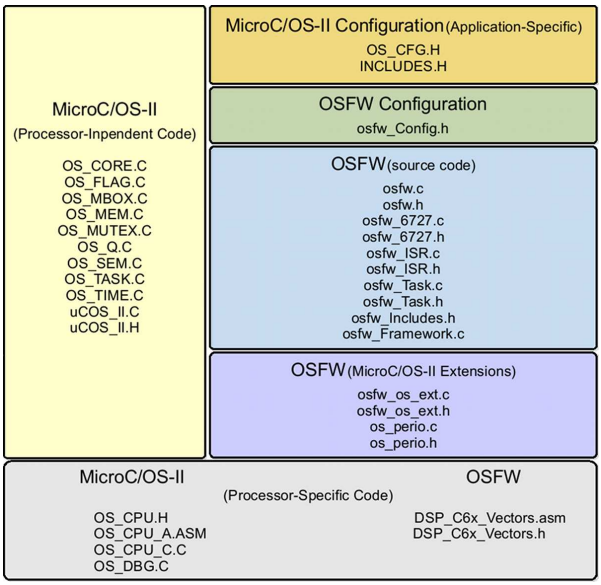

Figure 4.3: Arrangement of the source code files of µCOS-II and the OSFW.

5 Analysis of Results

The most important result of the project in current development phase is that the OSFW provides an extended RTOS platform, which can be utilised as a groundwork in divergent digital control applications. The OSFW has been developed, tested, and executed on the two separate EDC-DSP boards and the results are encouraging as the program operates as expected. Therefore, operational status of the OSFW is examined via memory consumption and CPU utilisation metric as it's considered to be to the most important marker concerning the OSFW's efficiency and overall performance. The framework's memory usage is also a momentous figure as it depicts relations between the OSFW, µC/OS-II, and the DSP related libraries and patches in detailed manner. Generally, the current configuration of the board provides enough external memory for tasking applications, thus not inducing any memory restrictions concerning the OSFW utilisation. However, microcode update patches issued into the DSP, and requirements for additional support libraries claim significant amount of memory.

5.1 Benchmark Tests

The selected benchmarks; CPU utilisation and memory consumpition depicts well, how the system's resources are used. Generally, none of the markers are close to theoretical or practical limits, thus demonstrating the OSFW's capability to answer to predefined real-time computing requirements and hardware related boundaries. The CPU utilisation test is carried out with an internal statistics task of the RTOS, which gives CPU usage as a percentage value (0–100%) with a resolution of one percent. Memory usage is a calculated value from output of the linker's memory map, thus depicting actual memory requirements of the OSFW's machine code. In addition, the OSFW's system tasks stack memory requirements are measured with µC/OS-II's stack checking tool.

5.1.1 CPU Utilisation

The OSFW's CPU utilisation percentage is composed of running system tasks and an application task. Generally, all system tasks lay in ready state as the frameworks API services are not used during the tests, excluding the watch dog task that executes after a periodic fashion at one system tick interval (100 µs). Hence, the measurement depicts a situation, where a simple periodic application task is running without user interaction. The system's RAM is used for test as it's the fastest memory type available for the OSFW. CPU utilisation shows clearly that the TMS320C6727 DSP of the EDC-DSP board is very efficient and provides lots of computing power at the OSFW's disposal. Commonly, the results indicate that optimization has little effect on total CPU usage. Nonetheless, CPU usage reduces with higher optimization settings, thus pointing out a clear trend how the settings affect on program code execution. In Table 5.1 is presented the results of program code optimization.

Table 5.1: Optimization of the OSFW: Effects of the CCS compiler's options on CPU usage.

OSFW Optimization	Optimization switches				OSFW Functioning	
	Size	Speed	Opt. Level	Debug	CPU Usage [%]	State
Debug	0	0	none	Full	1	ok
	0	5	Register (-o0)	Full	1	ok
	0	5	Local (-o1)	Full	<1	ok
	0	5	Function (-o2)	Full	<1	ok
	0	5	File (-o3)	Full	<1	ok
Release	1	5	Register (-o0)	None	1	ok
	1	5	Local (-o1)	None	$\leq$1	ok
	1	5	Function (-o2)	None	<1	ok
	1	5	File (-o3)	None	<1	ok

When the results are compared with theoretical CPU utilisation values in Table 3.1, it's axiomatic that the OSFW fits the bill of the set real-time software execution requirements. The watch dog causes presumably the most significant CPU utilisation within the OSFW as it's executed in phase with the system's tick clock. In addition, context switching at the selected tick rate produces extra burden on the DSP. Commonly, the OSFW lefts 98.6 percent of theoretically available CPU time at a control application's disposal.

5.1.2 Memory Usage

Memory consumption review is performed within RAM memory as the DSP's on-chip RAM provides enough memory capacity for the OSFW and the additional DSP related software libraries. The usage is calculated from the release domain executable, which is compiled with the maximum performance optimization options. However, the size optimization is left rather undemanding as performance of code suffers from excessive size optimization. Generally, the most performance critical code should be compiled without size optimization [26, p. 63]. Concerning the EDC-DSP board, program code of the OSFW can be located into SDRAM upon external boot up procedure, thus freeing space in RAM, if needed. The downside is that access speed of SDRAM is three times slower compared to RAM. In Table 5.2 is presented memory consumption of the OSFW, µC/OS-II, and test application.

Table 5.2: Memory consumption the OSFW, DSP resources, and application task in the system's RAM memory.

Code Sections	RAM Memory Consumption			
	Size [Hex]	Size [KB]	Usage [%]	Notes
OSFW	0x00005C56	23.0	9.3	All of the OSFW's code
µC/OS-II	0x000090AE	36.2	14.5	The RTOS: µC/OS-II
Test Task	0x00000918	2.3	0.9	Application task code
DSP related	0x0000877C	33.9	13.6	Patches, libraries, stack, etc.
Free Memory	0x00026667	153.6	61.7	Available free memory
Available RAM	0x0003E3FF	249.0	100.0	Defined within the configuration file of the linker

The results shows that the OSFW's program code, including the RTOS, takes up 60 KB of memory, which is 24 percent of all available RAM memory. Hence, it's obvious that at some point, as the software evolves, SDRAM will become principal memory location for the OSFW's. In addition, when comparing sizes of the OSFW and µC/OS-II, it's noticeable that the full featured OSFW is merely 64 percent of size of µC/OS-II. Commonly, any control applications that utilises the current version of the OSFW, will probably take approximately 24–35 percent of available RAM memory. Therefore, the EDC-DSP board's memory resources are well sized for the OSFW based applications.

5.1.2.1 Stack Sizes of the System Tasks

Memory requirements of the system tasks' stacks are checked with a `OSTaskStkChk()` function, which calculates a measure of memory needed by a task. The value isn't exact as the task's stack allocation demands may vary during program execution. Therefore, it advised that task's stack allocation value exceeds the reported value by factor 1.1–2, depending on amount of local variables [1, p. 125–129]. The current version of the OSFW utilises safety margin factor: 1.3 or above to limit RAM usage. Commonly, the system's tasks stack sizes should be checked and revised, if program code execution of the OSFW occurs from SDRAM or equivocal system execution errors occurs within the system. In Table 5.3 is presented stack allocations of the OSFW's system tasks.

Table 5.3: Stack size configuration of the OSFW's system tasks.

OSFW TASK LIST	Task Size Allocation		
	Stack Usage [B]	Set Stack Size [B]	Spare [%]
OSFW_TaskSPIErrHandler	384	512	33.3
OSFW_TaskErrHandler	336	448	33.3
OSFW_TaskCommRX	344	448	30.2
OSFW_TaskCommTX	344	448	30.2
OSFW_TaskCommandHandler	368	480	30.4
OSFW_TaskWatchdog	304	400	31.6

5.2 Aspects of the OSFW

The OSFW software composition including µC/OS-II with the DSP specific port and periodic extension is the principal result of the project. Commonly, the RTOS supports the TMS320C6727 DSP and provides interoperability with the EDC-DSP board's other processors; FPGA and FDSP. The OSFW is bootable in AIS binary format, thus the system can be utilised via the FDSP's bootloader utility. Commonly, the system is highly configurable and efficient hard real-time environment as behavior and configuration of the system can be altered extensively with the configuration options. The system embodies transparent wall-to-wall hardware abstraction layer within the system APIs, thus promoting the information hiding principle. Modular structure of the software and extensive documentation have already proved that the system's maintenance and further development are effectual as the OSFW is already being developed in a distributed manner.

The software is debugged and tested with the EDC-DSP board and functionality of the system is tested by the author and the staff of LCEDS on two independent EDC-DSP boards. Although the functioning of the framework is verified, the testing suffered from mysterious crashing of the JTAG adapter. Typically, the software could be loaded into the DSP via the JTAG only once, whereby sequential downloading attempt caused fuzzy system operation, which required to cut power from the adapter and the EDC-DSP board alike. Commonly, erroneous execution such as ghost value writes were perceived within the DSP memory registers. Another typical problem owing to the JTAG adapter was that the software's execution didn't advance even into `main()` function, while the IDE debugging tools showed that system was running. This was verified by utilising software and hardware breakpoints.

On extremely rare occasion, CPU utilisation calculation went haywire showing values from 0 to 127 (0x00–0x7F), thus illustrating all possible values of a signed 8-bit variable. The reason for this behavior is uncertain as the same binary file functioned perfectly after the adapter cold boot. It's very likely that usage of this particular adapter is broken or causes some interference every time when CPU is accessed as it's stopped momentarily by the adapter. Generally, it's suggested that CPU usage should be examined more closely, when the software has matured to production state and provides a functional interface via the FDSP. Commonly, the results are shown to be reliable and believable concerning high speed system tick utilisation with µC/OS-II as the results are in step with findings of the former study by Penttinen [16].

6 Conclusions

The purpose of the thesis is to find general use solution for control application development in a real-time operating system environment. A preliminary study is conducted to discover how a semantic service layer framework with divergent preemptive real-time operating systems can be utilised with a dedicated hardware in a versatile manner. A real-time operating system framework is designed and programmed as a resolution to set research subjects. In addition, an extensive documentation is produced to support maintenance of the software and potential further development. The requirements for the system are set in accordance with active magnet bearing controlling standards, that is high speed control loop execution. The requirements are examined in detailed manner and discerned to be realisable. Elicited requirements for the system are:

- Communication and synchronisation capability between the FPGA and DSP of the EDC-DSP board.
- Resolution of secure dynamic memory allocation concerning internal and external memories.
- Deterministic hard real-time computing of periodically executed control algorithms.
- Compliance with the Information hiding principle, while providing transparent API and HAL layers.
- Structural composition of a skeletal software structure that allows increment of external software components.
- Thorough documentation that promotes upkeep and further development of the software in multi-developer environment.

6.1 Outcome of the Project

The research objectives are met by designing a semantic framework application that utilises the real-time operating system MicroC/OS-II and the periodic extension designed for the RTOS. Therefore, the OSFW incorporates code portions of the framework, µC/OS-II, and the periodic extension, thus making control application development convenient as the OSFW provides hardware abstraction layer for the EDC-DSP board's hardware and various system APIs at a developer's disposal. The corollary for software development issues are presented in Chapters 3 and 4 and are as follows:

- Communication problem between the FPGA and DSP is solved with effectual management of non-maskable interrupt that is used for the purpose. In addition, two synchronisation methods are provided with a dedicated low level hardware driver to perform data exchange between the processors.
- All dynamic memory allocation and management is carried out via µC/OS-II memory services. The standard C memory allocation functions are not used.
- Periodic extension is used to provide deterministic program execution and hard real-time operation is achieved by careful software design and programming, thus the number of system tasks is kept reasonable to avoid complex task code.
- All hardware is subtly obscured from a user as application program interface services provide detailed and thorough access into the DSP's hardware.
- The OSFW is divided into semantic system APIs, hardware layer, and base structure, whereby similar apportionment is extended to source code files. In addition, configuration options are used to alter the structure of the OSFW as software blocks can be included or excluded in accordance to the options.
- The source code of the OSFW is augmented with inlined documentation, which is prosessed with an external open source code documentation tool: Doxygen. Hence, the documentation is generated automatically from the OSFW's source codes, thus upkeeping the software and ensuring that changes done upon further development are preserved in the documentation.

The main contribution of the project is that the preemptive hard real-time operating system framework, which is able to execute at very high tick rate, is produced as a proposal for demanding control applications requirements. The OSFW framework, excluding the utilised RTOS and the periodic extension, is designed and programmed from the start, thus revealing the application framework that is versatile; the standard C compliant (C89), highly portable, and loosely tied to the underlying real-time operating system software. Concerning the EDC-DSP board's performance with the OSFW, the results are considered to be credible as repetitive software tests didn't expose any unexplainable anomalies in the software's functionality.

The OSFW is tested thoroughly but the FPGA data driver, FPGA's flash memory initialisation and communication interface between the DSP and FDSP are not tested, because the FPGA and FDSP are lacking required software to make the testing possible, thus leaving express issues to be solved in further development. In addition, the programming was done on the author's computer, thus preventing usage of a dedicated Subversion version control system server of the LCEDS as that part of the network of Lappeenranta University of Technology cannot be accessed from the internet. It's fair to say that all issues concerning the research theme are resolved as all essential parts of the OSFW software are fully functional. Hence, the software already provides suitable development platform for tasking control algorithms. However, the OSFW is not complete, which is denoted with the framework's current version number (0.9.0). Therefore, additional development is needed to finalize the OSFW.

6.2 Future Prospects

Additional development is certainly needed concerning the EDC-DSP environment as the FPGA data driver's synchronization needs to be verified and SPI communication driver needs to be tested. Additionally, functions for some algorithms such a cyclic redundancy check (CRC) are programmed and intentionally left unfinished as a place holder for further development. These functions were never part of the research plan, thus the purpose is give guidelines and ideas, where the software design could be advancing. One of the prospective enhancements that should be considered concerning the OSFW memory API, is that the standard C memory copy function: `memcpy()`, which is used by the OSFW's memory copy function, should be rewritten from the start as an integral part of the OSFW, thus attenuating dependence on external software.

Since the research plan's inception; the MESCHACH, a matrix calculus software library by David Stewart and Zbigniew Leyk of the Australian National University, was going to be added into the OSFW. The matrix library inclusion was eliminated as a result of already enlarged project and reorganisation of the thesis. However, thus far the matrix library has been successfully tested with the OSFW.

Virtually all previous research projects of the LCEDS, which have utilised the EDC-DSP board, concentrated on the FPGA utilisation as the DSP were left more or less in the wilderness. Hence, the OSFW opens up new possibilities how the board can be utilised in novel control applications. Generally, the future looks bright for the OSFW as additional development is already in the wind. Therefore, the OSFW has all the potential to become versatile real-time operating system framework for exacting control system applications.

References

[1] Jean J. Labrosse. *MicroC/OS-II The Real-Time Kernel*. CMP Books, second edition, 2002, ISBN 1-57820-103-9.

[2] Julius Luukko. *Sulautetut prosessorijärjestelmät [Embedded Processor Systems]*. LTY/Digipaino, study material copy (in finnish) edition, 2005, page: 57.

[3] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall PRT, Englewood Cliffs, New Jersey, second edition, 1988, ISBN 0-13-110362-8 pages: 185–189.

[4] Julius Luukko. Periodic Task Execution in µC/OS-II. Documentation of periodic extension for µC/OS-II, October 2003.

[5] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[6] Digital Signal Processors Silicon Errata. Data sheet number SPRZ232F, July 2005 http://focus.ti.com/lit/er/sprz232f/sprz232f.pdf.

[7] Jakub Nowack. LUT2007 Project, Configuring Code Composer Studio. December 2007.

[8] TMS320C6727, TMS320C6726, TMS320C6722 Floating-Point Digital Signal Processors. Data sheet number SPRS268E, January 2007 http://focus.ti.com/lit/ds/sprs268e/sprs268e.pdf.

[9] TMS320C672x DSP Real-Time Interrupt Reference Guide. Data sheet number SPRU717, April 2005 http://focus.ti.com/lit/ug/spru717/spru717.pdf.

[10] TMS320C672x DSP Dual Data Movement Accelerator (dMAX) Reference Guide. Data sheet number SPRU795D, October 2007 http://focus.ti.com/lit/ug/spru795d/spru795d.pdf.

[11] TMS320C672x DSP External Memory Interface (EMIF) User's Guide. Data sheet number SPRU711C, April 2007, pages: 9–11, 49–64
http://focus.ti.com/lit/ug/spru711c/spru711c.pdf.

[12] TMS320F2809, TMS320F2808, TMS320F2806, TMS320F2802, TMS320F2801, TMS320C2802, TMS320C2801, and TMS320F2801x DSPs Data Manual. Data sheet number SPRS230J, July 2007
http://focus.ti.com/lit/ds/sprs230j/sprs230j.pdf.

[13] Virtex-4 Family Overview. Data sheet number DS112 (v3.0), September 2007, page: 4, http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf.

[14] Phillip A. Laplante. *Real-Time Systems Design and Analysis*. IEEE press, 3rd edition, 2004, ISBN 0-471-22855-9.

[15] Institute of Electrical and Electronics Engineers. Recommended Practice for Software Requirements Specification. *IEEE std 830-1998*, 1998, ISBN 0-7381-0332-2, pages: 9–10.

[16] Aki Penttinen. FPGA:lle sulautetulla mikroprosessorilla toteutettu sähkökäytön säätöjärjestelmä [Control system for electric drives implemented with FPGA embedded microprocessor] (in Finnish). Master's thesis, Lappeenrannan teknillinen yliopisto, May 2005, pages: 66–77.

[17] B. Westergren L. Råde. *Mathematics Handbook for Science and Engineering*. Studentlitteratur, Lund, 4th edition, 2006, ISBN 91-44-00839-2.

[18] Dimitri van Heesch. Doxygen Manual for Version 1.5.3, 2007, pages: 15–43
http://www.doxygen.org.

[19] M. Bond and R. Anderson. API-level attacks on embedded systems. *Computer*, 34(10):67–75, 2001.

[20] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov. Cryptographic Processors - A Survey. *Proceedings of the IEEE*, 92(2):357–369, February 2006.

[21] D. L. Parnas. Designing Software for Ease of Extension and Contradiction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, March 1979.

[22] J. Salli. RTOS Framework for Real-Time Control Systems - Reference Manual. August 2009.

[23] TMS320C6000 DSP/BIOS Users's Guide. Data sheet number: SPRU303B, May 2000, pages: 3-2 – 3-30, http://focus.ti.com/lit/ug/spru303b/spru303b.pdf.

[24] D. Dart and S. Dirksen. Using the DSP/BIOS Kernel in Real-Time DSP Applications. Application Report SPRA781, August 2001, page: 7 http://focus.ti.com/lit/an/spra781/spra781.pdf.

[25] M. Richardson. CSL 2.x to CSL 3.x Migration. Application Report SPRAA10, January 2006, http://focus.ti.com/lit/an/spraa10/spraa10.pdf.

[26] TMS320C6000 Optimizing Compiler v6.1 User's Guide. Data sheet number SPRU187O, May 2008 http://focus.ti.com/lit/ug/spru187o/spru187o.pdf.

[27] TMS320C672x DSP Serial Peripheral Interface (SPI) Reference Guide. Data sheet number SPRU718B, July 2007, pages: 29–31 http://focus.ti.com/lit/ug/spru718b/spru718b.pdf.

[28] TMS320C672x DSP Inter-Integrated Circuit (I2C) Module Reference Guide. Data sheet number SPRU877E, December 2007, pages: 23–36 http://focus.ti.com/lit/ug/spru877e/spru877e.pdf.

[29] Richard M. Stallman. Using the GNU Compiler Collection for GCC Version 4.1.2. GNU Press, 2005, http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc.pdf.

[30] Valgrind Documentation Release 3.4.0, January 2009 http://valgrind.org/docs/manual/valgrind_manual.pdf.

[31] Andreas Zeller. Debugging with DDD User's Guide and Reference Manual, January 2004, pages: 5–35, 92–139 http://www.gnu.org/manual/ddd/pdf/ddd.pdf.

[32] TMS320C6000 Code Composer Studio Tutorial. Data sheet number SPRU301C, February 2000, pages:1-1 – 1-18, 4-1 – 4-4 http://focus.ti.com/lit/ug/spru301c/spru301c.pdf.

[33] Karen Baldwin. Using the TMS320C672x Bootloader. Application Report SPRAA69C, March 2006, http://focus.ti.com/lit/an/spraa69c/spraa69c.pdf.

Decision Table of RTOS Selection

Criterion	Specification	w_i	DSP/BIOS [m_i]	µC/OS-II [m_i]
m_1	Minimum Interrupt latency	1.00	1.00	0.95
m_2	Task management	0.75	0.60	0.50
m_3	Memory requirements	0.80	0.90	1.00
m_4	Scheduling mechanism	0.50	0.75	0.50
m_5	Intertask communication methods	1.00	1.00	1.00
m_6	Product life cycle support	0.50	1.00	0.60
m_7	Availability of supporting software	1.00	1.00	0.50
m_8	Hardware compatibility	1.00	0.10	1.00
m_9	Availability of source code	1.00	0.25	1.00
m_{10}	Context switch time	0.90	1.00	1.00
m_{11}	Price of software	0.75	0.50	1.00
m_{12}	Availability of development platforms	1.00	0.10	0.90
m_{13}	Supported network protocols	0.25	0.50	0.00
M	Fitness metric		6.90	8.83

The fitness criterion metric is calculated as

$$M = \sum_{i=1}^{13} w_i m_i \tag{AI.1}$$

DSP/BIOS fitness metric

$$\begin{aligned} M = {} & (1.00 \cdot 1.00) + (0.75 \cdot 0.60) + (0.80 \cdot 0.90) + (0.50 \cdot 0.75) \\ & + (1.00 \cdot 1.00) + (0.50 \cdot 1.00) + (1.00 \cdot 1.00) + (1.00 \cdot 0.10) \\ & + (1.00 \cdot 0.25) + (0.90 \cdot 1.00) + (0.75 \cdot 0.50) + (1.00 \cdot 0.10) \\ & + (0.25 \cdot 0.50) = 6.90 \end{aligned} \tag{AI.2}$$

µC/OS-II fitness metric

$$\begin{aligned} M = {} & (1.00 \cdot 0.95) + (0.75 \cdot 0.50) + (0.80 \cdot 1.00) + (0.50 \cdot 0.50) \\ & + (1.00 \cdot 1.00) + (0.50 \cdot 0.60) + (1.00 \cdot 0.50) + (1.00 \cdot 1.00) \\ & + (1.00 \cdot 1.00) + (0.90 \cdot 1.00) + (0.75 \cdot 1.00) + (1.00 \cdot 0.90) \\ & + (0.25 \cdot 0.00) = 8.83 \end{aligned} \tag{AI.3}$$

Configuration File of the MicroC/OS-II for the OSFW

```
/*
*********************************************************************************************************
*                                              uC/OS-II
*                                        The Real-Time Kernel
*
*                           (c) Copyright 1992-2003, Jean J. Labrosse, Weston, FL
*                                           All Rights Reserved
*
*                                  uC/OS-II Configuration File for V2.7x
*
* File : OS_CFG.H
* By   : Jean J. Labrosse
*
* Revision history:  Modifications to TI C672x by Julius Luukko, 2007
*        Modifications to OSFW by Juha Salli 2009
*********************************************************************************************************
*/

#ifndef OS_CFG_H
#define OS_CFG_H

/** C672x port specific defines ( Partially defined under different name in the RTI section of the osfw_cC6727.h ) */

/* RTI configuration: CPU clock frequency */
#define CPU_CLOCK_HZ 300000000

/** RTI configuration: Compare timer interrupt frequency. */
#define RTI_CLOCK_HZ 10000

/**RTI configuration: Compare limit */
#define RTI_COMPARE0_LIMIT 1
                                    /* ---------------------- MISCELLANEOUS ----------------------- */
#define OS_ARG_CHK_EN             1    /* Enable (1) or Disable (0) argument checking                  */
#define OS_CPU_HOOKS_EN           0    /* uC/OS-II hooks are found in the processor port files         */

#define OS_DEBUG_EN               1    /* Enable(1) debug variables                                    */

#define OS_EVENT_NAME_SIZE       32    /* Determine the size of the name of a Sem, Mutex, Mbox or Q    */

#define OS_LOWEST_PRIO           63    /* Defines the lowest priority that can be assigned ...         */
                                       /* ... MUST NEVER be higher than 63!                            */

#define OS_MAX_EVENTS            20    /* Max. number of event control blocks in your application      */
#define OS_MAX_FLAGS              5    /* Max. number of Event Flag Groups    in your application      */
#define OS_MAX_MEM_PART          17    /* Max. number of memory partitions                             */
#define OS_MAX_QS                10    /* Max. number of queue control blocks in your application      */
#define OS_MAX_TASKS             15    /* Max. number of tasks in your application, MUST be >= 2       */

#define OS_SCHED_LOCK_EN          1    /*     Include code for OSSchedLock() and OSSchedUnlock()       */

#define OS_TASK_IDLE_STK_SIZE   128    /* Idle task stack size (# of OS_STK wide entries)              */

#define OS_TASK_STAT_EN           1    /* Enable (1) or Disable(0) the statistics task                 */
#define OS_TASK_STAT_STK_SIZE   128    /* Statistics task stack size (# of OS_STK wide entries)        */
#define OS_TASK_STAT_STK_CHK_EN   1    /* Check task stacks from statistic task                        */

#define OS_TICK_STEP_EN           0    /* Enable tick stepping feature for uC/OS-View                  */
#define OS_TICKS_PER_SEC (((CPU_CLOCK_HZ/2)/RTI_CLOCK_HZ-1)/RTI_COMPARE0_LIMIT)


                                    /* ----------------------- EVENT FLAGS ------------------------- */
#define OS_FLAG_EN                1    /* Enable (1) or Disable (0) code generation for EVENT FLAGS    */
#define OS_FLAG_WAIT_CLR_EN       1    /* Include code for Wait on Clear EVENT FLAGS                   */
#define OS_FLAG_ACCEPT_EN         1    /*     Include code for OSFlagAccept()                          */
#define OS_FLAG_DEL_EN            1    /*     Include code for OSFlagDel()                             */
#define OS_FLAG_NAME_SIZE        32    /*     Determine the size of the name of an event flag group    */
#define OS_FLAG_QUERY_EN          1    /*     Include code for OSFlagQuery()                           */


                                    /* -------------------- MESSAGE MAILBOXES ---------------------- */
#define OS_MBOX_EN                1    /* Enable (1) or Disable (0) code generation for MAILBOXES      */
#define OS_MBOX_ACCEPT_EN         1    /*     Include code for OSMboxAccept()                          */
#define OS_MBOX_DEL_EN            1    /*     Include code for OSMboxDel()                             */
```

```
#define OS_MBOX_POST_EN          1    /*     Include code for OSMboxPost()                          */
#define OS_MBOX_POST_OPT_EN      1    /*     Include code for OSMboxPostOpt()                       */
#define OS_MBOX_QUERY_EN         1    /*     Include code for OSMboxQuery()                         */


                                      /* --------------------- MEMORY MANAGEMENT -------------------- */
#define OS_MEM_EN                1    /* Enable (1) or Disable (0) code generation for MEMORY MANAGER */
#define OS_MEM_QUERY_EN          1    /*     Include code for OSMemQuery()                          */
#define OS_MEM_NAME_SIZE        32    /*     Determine the size of a memory partition name          */


                                      /* ---------------- MUTUAL EXCLUSION SEMAPHORES --------------- */
#define OS_MUTEX_EN              1    /* Enable (1) or Disable (0) code generation for MUTEX          */
#define OS_MUTEX_ACCEPT_EN       1    /*     Include code for OSMutexAccept()                       */
#define OS_MUTEX_DEL_EN          1    /*     Include code for OSMutexDel()                          */
#define OS_MUTEX_QUERY_EN        1    /*     Include code for OSMutexQuery()                        */


                                      /* ---------------------- MESSAGE QUEUES ---------------------- */
#define OS_Q_EN                  1    /* Enable (1) or Disable (0) code generation for QUEUES         */
#define OS_Q_ACCEPT_EN           1    /*     Include code for OSQAccept()                           */
#define OS_Q_DEL_EN              1    /*     Include code for OSQDel()                              */
#define OS_Q_FLUSH_EN            1    /*     Include code for OSQFlush()                            */
#define OS_Q_POST_EN             1    /*     Include code for OSQPost()                             */
#define OS_Q_POST_FRONT_EN       1    /*     Include code for OSQPostFront()                        */
#define OS_Q_POST_OPT_EN         1    /*     Include code for OSQPostOpt()                          */
#define OS_Q_QUERY_EN            1    /*     Include code for OSQQuery()                            */


                                      /* ------------------------ SEMAPHORES ------------------------ */
#define OS_SEM_EN                1    /* Enable (1) or Disable (0) code generation for SEMAPHORES     */
#define OS_SEM_ACCEPT_EN         1    /*    Include code for OSSemAccept()                          */
#define OS_SEM_DEL_EN            1    /*    Include code for OSSemDel()                             */
#define OS_SEM_QUERY_EN          1    /*    Include code for OSSemQuery()                           */


                                      /* --------------------- TASK MANAGEMENT ---------------------- */
#define OS_TASK_CHANGE_PRIO_EN   1    /*     Include code for OSTaskChangePrio()                    */
#define OS_TASK_CREATE_EN        1    /*     Include code for OSTaskCreate()                        */
#define OS_TASK_CREATE_EXT_EN    1    /*     Include code for OSTaskCreateExt()                     */
#define OS_TASK_DEL_EN           1    /*     Include code for OSTaskDel()                           */
#define OS_TASK_NAME_SIZE       32    /*     Determine the size of a task name                      */
#define OS_TASK_PROFILE_EN       1    /*     Include variables in OS_TCB for profiling              */
#define OS_TASK_QUERY_EN         1    /*     Include code for OSTaskQuery()                         */
#define OS_TASK_SUSPEND_EN       1    /*     Include code for OSTaskSuspend() and OSTaskResume()    */
#define OS_TASK_SW_HOOK_EN       1    /*     Include code for OSTaskSwHook()                        */


                                      /* --------------------- TIME MANAGEMENT ---------------------- */
#define OS_TIME_DLY_HMSM_EN      1    /*     Include code for OSTimeDlyHMSM()                       */
#define OS_TIME_DLY_RESUME_EN    1    /*     Include code for OSTimeDlyResume()                     */
#define OS_TIME_GET_SET_EN       1    /*     Include code for OSTimeGet() and OSTimeSet()           */
#define OS_TIME_TICK_HOOK_EN     1    /*     Include code for OSTimeTickHook()                      */

typedef INT16U             OS_FLAGS;  /* Date type for event flag bits (8, 16 or 32 bits)           */

#endif
```

The CCS's Linker configuration File for the OSFW

```
/*

RTOS Framework for Real-Time Control System
=======================================

Author: Juha Salli

Lappeenranta University of Technology (LUT)
Faculty of Technology
Laboratory of Digital and Control Engineering

Copyright Lappeenranta University of Technology (LUT), All rights reserved

Linker command version : 0.9.0

*/
/* < Linker Configuration Swiches >*/
--rom_model

   CCSPatches\applypatch.obj
-l CCSPatches\c672xrompatchv1_00_00.lib

//   CCSPatches\applySystempatch.obj
//-l CCSPatches\c672xSystemPatchV2_00_00.lib

-l rts67plus.lib

/* < Memory Map > */

MEMORY
{
  iROM_BOOTLOADER (RX): o = 0x00000000 l = 0x1FFFF  /* Internal ROM: ROM Bootloader code  */
  iROM_DSPLIB     (RX): o = 0x00020000 l = 0xBFFF   /* DSP library*/
  iROM_FastRTSLIB (RX): o = 0x0002C000 l = 0x3FFF   /* Fast RST library*/
  iROM_DSPBIOS    (RX): o = 0x00030000 l = 0x2FFFF  /* DSP BIOS memory */

  iRAM_BOOTLOADER       o = 0x10000000 l = 0x0FFF   /* Reserved at Boot for Bootloader*/
  iRAM_DSPBIOS          o = 0x10001000 l = 0x0AFF   /* Reserved for DSP/BIOS */
  iRAM_FastRTS          o = 0x10001B00 l = 0x00FF   /* Reserved for fast RTS */

  iRAM_OSFW             o = 0x10001C00 l = 0x00025800  /* ..for the OSFW and user applications  150KB*/
  iRAM_FREE             o = 0x10027400 l = 0x00018BFF  /* .. for misc. usage (dynamic allocation etc). ~99KB */
  SDRAM_OSFW            o = 0x802EE000 l = 0x000F9FFF  /* External SDRAM: 1 MB */
}


/* < Section Allocation > */

/* Refer to "TMS320C6000 Optimizing Compiler v6.1 User's Guide" (SPRU187O) */

SECTIONS
{
    VECTORS     >  iRAM_OSFW
    .TIBoot:    >  iRAM_OSFW
    .vectors    >  iRAM_OSFW /*""*/
    .cinit      >  iRAM_OSFW /*"Tables for explicitly initialized global and static variables."*/
    .pinit      >  iRAM_OSFW /*"Table for calling global object constructors at run time".*/
    .text       >  iRAM_OSFW /*"Executable code and constants"*/
    .stack      >  iRAM_OSFW
    .bss        >  iRAM_OSFW /*"Global and static variables"*/
    .const      >  iRAM_OSFW /*".const contains string literals, floating-point constants, and const"*/
    .far        >  iRAM_OSFW /*".far reserves space for global and static variables that are declared far."*/
    .switch     >  iRAM_OSFW /*"Jump tables for large switch statements"*/
    .sysmem     >  iRAM_OSFW /*".sysmem reserves space for dynamic memory allocation" Not used in the OSFW */
    .cio        >  iRAM_OSFW /*""*/
    .data       >  iRAM_OSFW /*".data section is an initialized section that contains initialised data."*/
    .tables     >  iRAM_OSFW
}
```

CPU Interrupt Handling Functions and Associated Extension Hooks

```
_Disable_int:

  CALL    _OSEnterCriticalHook;
  NOP    5;
  mvc    CSR, B4
  and    1, B4, B0
        [!B0] CLR B4, 1 ,1 ,B4
        [B0] SET B4, 1, 1, B4
  CLR    B4, 0, 0, B4
  mvc    B4, CSR
  B      B3
  NOP    5;


_Enable_int: ;

  CALL    _OSExitCriticalHook;
  NOP    5;
  mvc    CSR, B4
  and    2, B4, B0
        [!B0] CLR B4,0,0,B4
        [B0] SET B4,0,0,B4
  mvc    B4, CSR
  B      B3
  NOP    5;


extern void OSEnterCriticalHook(void)
{
  OSFW_FPGA_IRQ_Disable;
}


extern void OSExitCriticalHook(void)
{
  OSFW_FPGA_IRQ_Enable;
}
```

Source Code of the Watchdog Task with Doxygen Commentary

```
/** \b Description
\n This function is the framework's watchdog task. The task runs
in a periodic fashion and resets the hardware watchdog timer on
every program cycle. If the OSFW or RTOS freezes because of a soft-
ware bug, the hardware watchdog timer will reset the system af-
ter the preset delay. Refer to the documentation of the watchdog
control function OSFW_C6727_Watchdog(). The task uses the perio-
dic uC/OS-II extension by courtesy of J. Luukko. The task will
run only if the compliler directive OSFW_DRCTV_WATCHDOG_EN
is set.

\n <b>A developer note:</b> A periodic cycle delay parameter
should equal or faster than the watchdog timer.

\b Arguments
\n None

<b> Return Value </b>
\n None

<b> Pre Condition </b>
\n Initialised OSFW environment
\n Periodic uC/OS-II extension

<b> Post Condition </b>
\n The hardware watchdog timer is reset.

\b Modifies
\n Real-time interrupt register (RTIWDKEY)
*/


void OSFW_TaskWatchdog(void *pdata)
{
     OSTaskMakePeriodic(OS_PRIO_SELF, 1);
     while(1)
     {
          OSFW_C6727_Watchdog(RESETTIMER);
          OSTaskWaitPeriodic();
     }
}
```

Configuration File of the OSFW

```
/*

RTOS Framework for Real-Time Control Systems
====================================

Author: Juha Salli

Faculty of Technology, Laboratory of Digital and Control Engineering
Copyright Lappeenranta University of Technology (LUT)
All rights reserved

*/

/**
\file  osfw_Config.h

\n Description
\li A header file of the OSFW's compiler directives.

\n Modification
\li Source code version : 0.9.0
\li Source code date : 05-04-2009

\author Juha Salli
*/

#ifndef OSFW_CONFIG_H
#define OSFW_CONFIG_H

/*< Memory related options: > */

/** Initialise (clear) allocated SDRAM memory at boot. */
#define OSFW_DRCTV_CLR_SDRAM 1

/** Initialise (clear ) allocated FPGA (flash) memory at boot. */
#define OSFW_DRCTV_CLR_FPGA 0

/** Allocate free SDRAM memory to be used as general 'mallocable'
memory. */
#define OSFW_DRCTV_SDRAM_FREE_MALLOC 1

/** Initialise OS_AMB_InputData and OS_AMB_OutputData memory
```

```
blocks at boot. */
#define OSFW_DRCTV_CLR_AMB_AT_BOOT 1

/** Allocate SDRAM memory for ring buffer objects. */
#define OSFW_DRCTV_SDRAM_RINGBUF_MALLOC 1

/** Allocate memory for the data buffer used by ring-buffer objects.*/
#define OSFW_DRCTV_SDRAM_DATABUF_MALLOC 1

/** Allocate additional memory block sizes for dynamic memory
reservation. */
#define OSFW_DRCTV_SDRAM_MXXX_MALLOC 1


/* < Error handling related options: > */

/** Enable error handling in system tasks. */
#define OSFW_DRCTV_ERR_TASK_EN 1

/** Enable error handling in the memory API functions. */
#define OSFW_DRCTV_ERR_MEMAPI_EN 1

/** Enable error handling. */
#define OSFW_DRCTV_ERR_FRAMEFUNC 1

/** Enable printing of function names as text in the error handler
task. */
#define OSFW_DRCTV_ERR_FUNCNAME_EN 1

/** Enable error text generation in the error handler task. */
#define OSFW_DRCTV_ERR_PRINT_EN 1


/* < Task related options: > */

/** Put external I/O objects into SDRAM memory instead of RAM. */
#define OSFW_DRCTV_IO_VAR_IN_SDRAM 1

/** Enable watchdog */
#define OSFW_DRCTV_WATCHDOG_EN 1

/** Enable continuous EMIF NMI interrupt generation */
#define OSFW_DRCTV_EMIF_SYNC_CONT_EN 0
```

```
/* < Compiler related options: > */

/* If compiler fails to initialise pointers etc.
   (This directive enables additional initialisation code.) */
#define OSFW_DRCTV_COMPILER_INIT_FAIL 1


/* < Test environment related options:  >*/

/** JTAG environment is used to program DSP.*/
#define OSFW_DRCTV_JTAG_EN 1

/*
 See source code of AMB_example.c
*/

#endif

/* EOF */
```

Excerpt from the OSFW's documentation: Watch Dog task

11.10.2.7 void OSFW_TaskWatchdog (void * pdata)

Description

This function is the framework's watchdog task. The task runs in a periodic fashion and resets the hardware watchdog timer on every program cycle. If the OSFW or RTOS freezes because of a software bug, the hardware watchdog timer will reset the system after the preset delay. Refer to the documentation of the watchdog control function OSFW_C6727_Watchdog(). The task uses the periodic uC/OS-II extension by courtesy of J. Luukko. The task is will run only if the compiler directive OSFW_DRCTV_WATCHDOG_EN is set.

A developer note: A periodic cycle delay parameter should equal or faster than the watchdog timer.

Arguments

None

Return Value

None

Pre Condition

Initialised OSFW environment

Periodic uC/OS-II extension

Post Condition

The hardware watchdog timer is reset.

Modifies

Real-time interrupt register (RTIWDKEY)

Definition at line 577 of file osfw_Task.c.

Output of Valgrind’s Memory Checker Tool

```
==17697== Memcheck, a memory error detector.
==17697== Copyright (C) 2002-2008, and GNU GPL’d, by Julian Seward et al.
==17697== Using LibVEX rev 1878, a library for dynamic binary translation.
==17697== Copyright (C) 2004-2008, and GNU GPL’d, by OpenWorks LLP.
==17697== Using valgrind-3.4.0, a dynamic binary instrumentation framework.
==17697== Copyright (C) 2000-2008, and GNU GPL’d, by Julian Seward et al.
==17697== For more details, rerun with: -v
==17697==
==17697== Conditional jump or move depends on uninitialised value(s)
==17697==    at 0x400A3E8: (within /lib64/ld-2.9.so)
==17697==    by 0x40038D7: (within /lib64/ld-2.9.so)
==17697==    by 0x401309D: (within /lib64/ld-2.9.so)
==17697==    by 0x4002182: (within /lib64/ld-2.9.so)
==17697==    by 0x4000C47: (within /lib64/ld-2.9.so)
==17697==  Uninitialised value was created by a stack allocation
==17697==    at 0x400A2A0: (within /lib64/ld-2.9.so)
==17697==
==17697== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 0 from 0)
==17697== malloc/free: in use at exit: 0 bytes in 0 blocks.
==17697== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
==17697== For counts of detected errors, rerun with: -v
==17697== All heap blocks were freed -- no leaks are possible.
```

Error API Utilisation

OSFW FUNCTION LIST		Error Classification		
		Function ID	Error Code	Error Class
RTOS Port Files	DSP_C6x_Vectors.asm	n/a	n/a	n/a
	os_cpu_a.asm	n/a	n/a	n/a
	os_cpu_c.c	n/a	n/a	n/a
Interrupt Service Routines	EMIF_NMI_ISR()	n/a	n/a	n/a
	I2Cx_SPIx_ISR()	n/a	n/a	n/a
	dMAX_TCINT_ISR()	n/a	n/a	n/a
RTOS Extensions	OSEnterCriticalHook	n/a	n/a	n/a
	OSExitCriticalHook	n/a	n/a	n/a
Memory API	OSFW_MemCheck()	0	OSFW_ERR_MEM_ACCESS	3
	OSFW_MemCopy()	n/a	n/a	n/a
	OSFW_MemClear()	n/a	n/a	n/a
	OSFW_TestBit()	n/a	n/a	n/a
	OSFW_SetWordRegisterBit()	n/a	n/a	n/a
	OSFW_ResetWordRegisterBit()	n/a	n/a	n/a
	OSFW_SetByteRegisterBit()	n/a	n/a	n/a
	OSFW_ResetByteRegisterBit()	n/a	n/a	n/a
	OSFW_SetWordRegisterMask()	n/a	n/a	n/a
	OSFW_ResetWordRegisterMask()	n/a	n/a	n/a
	OSFW_SetRegister()	n/a	n/a	n/a
	OSFW_ClearInterrupt()	n/a	n/a	n/a
	OSFW_EnableInterrupt()	n/a	n/a	n/a
	OSFW_DisableInterrupt()	n/a	n/a	n/a
	OSFW_IntToFloat()	n/a	n/a	n/a
	OSFW_LongToFloat()	n/a	n/a	n/a
	OSFW_IntToDouble()	n/a	n/a	n/a
	OSFW_LongToDouble()	n/a	n/a	n/a
	OSFW_FloatToInt()	n/a	n/a	n/a
	OSFW_DoubleToInt()	n/a	n/a	n/a
	OSFW_FloatToLong()	n/a	n/a	n/a
	OSFW_DoubleToLong()	n/a	n/a	n/a
	OSFW_NopDelay()	n/a	n/a	n/a
	OSFW_C6727_Watchdog()	n/a	n/a	n/a
Error API	OSFW_ErrorQueueInit()	n/a	n/a	n/a
	OSFW_ErrorMsgGen()	n/a	n/a	n/a
	OSFW_ErrorReport()	n/a	n/a	n/a
	OSFW_TaskErrHandler()	n/a	n/a	n/a
	OSFW_TaskSPIErrHandler()	1	OSFW_ERR_SPI_COM_ERROR	2
	OSFW_TaskWatchdog()	n/a	n/a	n/a

OSFW FUNCTION LIST		Error Classification		
		Function ID	Error Code	Error Class
Communication API	OSFW_CommBufQueueInit()	n/a	n/a	n/a
	OSFW_EventFlagInit()	n/a	n/a	n/a
	OSFW_CommDataSend()	n/a	n/a	n/a
	OSFW_CommDataParser()	n/a	n/a	n/a
	OSFW_CRC()	n/a	n/a	n/a
	OSFW_RxHwDriver()	n/a	n/a	n/a
	OSFW_TxHwDriver()	n/a	n/a	n/a
	OSFW_TaskCommTX()	2	OSQPend errors	2
	OSFW_TaskCommRX()	3	OSQPend errors	2
	OSFW_FPGA_IOMemInit()	n/a	n/a	n/a
	OSFW_FPGA_IODriver()	n/a	n/a	n/a
Command API	OSFW_CommandQueueInit()	n/a	n/a	n/a
	OSFW_DataObjCreate()	n/a	n/a	n/a
	OSFW_CommandInterpreter()	n/a	n/a	n/a
	OSFW_TaskCommandHandler()	4	OSMutexPend errors	5
			OSFW_ERR_CMD_ACCESS	6
			OSFW_ERR_INVALID_CMD	6
			OSQPend errors	2
Buffer API	OSFW_RingBufferInit()	n/a	n/a	n/a
	OSFW_RingBufferCheck()	n/a	n/a	n/a
	OSFW_RingBufferAdd()	n/a	n/a	n/a
	OSFW_RingBufferClear()	n/a	n/a	n/a
	OSFW_RingBufferCreate()	n/a	n/a	n/a
	OSFW_RingBufferDelete()	n/a	n/a	n/a
C99 Substitutes	OSFW_printf()	5	OSMemGet errors	2
	OSFW_strtok_r()	6	OSFW_ERR_INVALID_STR	6

The OSFW Testing Record: System Initialisation

TEST RECORD: SYSTEM INITIALISATION		Software Checkup Matrix							
		Logic Step	Function Code	Error handling	Variables	Directives	Arguments	Logic	Opt.
DSP_C6x_SystemInit()		1.1	ok	n/a	n/a	n/a	n/a	ok	n/a
OSFW_C6727_PLL_Init()		1.2	ok	n/a	n/a	n/a	n/a	ok	ok
	Set the external memory into self-refresh mode	1.2.1	ok	n/a	n/a	n/a	n/a	ok	n/a
	Set PLL into bypass mode	1.2.2	ok	n/a	n/a	n/a	n/a	ok	n/a
	Wait 4 cycles for the slowest PLLOUT to settle	1.2.3	ok	n/a	n/a	n/a	n/a	ok	n/a
	Reset PLL	1.2.4	ok	n/a	n/a	n/a	n/a	ok	n/a
	Divider 0 (CLKIN) and multiplier setup	1.2.5	ok	n/a	ok	n/a	n/a	ok	n/a
	Divider 1: CPU and internal memory	1.2.6	ok	n/a	ok	n/a	n/a	ok	n/a
	Divider 2: dMAX and peripherals	1.2.7	ok	n/a	ok	n/a	n/a	ok	n/a
	Divider 3: EMIF	1.2.8	ok	n/a	ok	n/a	n/a	ok	n/a
	Reset BR2 bridge	1.2.9	ok	n/a	n/a	n/a	n/a	ok	n/a
	Perform clock aligment	1.2.10	ok	n/a	ok	n/a	n/a	ok	n/a
	Wait until alignment is done	1.2.11	ok	n/a	n/a	n/a	n/a	ok	n/a
	Wait until PLL is reset. (Assertion time >=125 ns)	1.2.12	ok	n/a	n/a	n/a	n/a	ok	n/a
	Take PLL out of reset	1.2.13	ok	n/a	n/a	n/a	n/a	ok	n/a
	Wait until PLL is stabilised. (>= 187.5 µs)	1.2.14	ok	n/a	n/a	n/a	n/a	ok	n/a
	Enable PLL	1.2.15	ok	n/a	n/a	n/a	n/a	ok	n/a
	Wait until input source is stable	1.2.16	ok	n/a	n/a	n/a	n/a	ok	n/a
	Enable BR2 bridge	1.2.17	ok	n/a	n/a	n/a	n/a	ok	n/a
	Reset the self-refresh mode	1.2.18	ok	n/a	n/a	n/a	n/a	ok	n/a

TEST RECORD: SYSTEM INITIALISATION		Software Checkup Matrix							
		Logic Step	Function Code	Error handling	Variables	Directives	Arguments	Logic	Opt.
OSFW_C6727_EMIF_Init()		1.3	ok	n/a	n/a	n/a	n/a	ok	n/a
	Asynchronous 1 Configuration Register (A1CR)	1.3.1	untested	untested	untested	untested	untested	untested	untested
	Asynchronous Wait Cycle Configuration Register (AWCCR)	1.3.2	untested	untested	untested	untested	untested	untested	untested
	SDRAM timing register	1.3.3	ok	n/a	ok	n/a	n/a	ok	n/a
	SDRAM selfresresh exit timing register	1.3.4	ok	n/a	n/a	n/a	n/a	ok	n/a
	SDRAM refresh control register	1.3.5	ok	n/a	n/a	n/a	n/a	ok	n/a
	SDRAM control register	1.3.6	ok	n/a	ok	n/a	n/a	ok	n/a
	Perform a test read to cause delay	1.3.7	ok	n/a	n/a	n/a	n/a	ok	n/a
OSFW_C6727_dMAX_Init()		1.4	ok	n/a	n/a	n/a	n/a	ok	n/a
	Set pointer to the SPI1 transfer table	1.4.1	ok	n/a	ok	n/a	n/a	ok	ok
	Event control setup	1.4.2	ok	n/a	n/a	n/a	ok	ok	n/a
	Event and Transfer tables setup	1.4.3	ok	n/a	ok	n/a	ok	ok	n/a
	Event signal edge setup	1.4.4	ok	n/a	n/a	n/a	ok	ok	n/a
	Enable SPI DMA transfer	1.4.5	ok	n/a	n/a	n/a	ok	ok	n/a
OSFW_C6727_I2C_Init()		1.5	ok	n/a	n/a	n/a	n/a	ok	n/a
	I2C 1 Prescaler Register setup	1.5.1	ok	n/a	n/a	n/a	ok	ok	n/a
	I2C 1 Clock registers setup	1.5.2	ok	n/a	n/a	n/a	ok	ok	n/a
	I2C 1 Mode register setup	1.5.3	ok	n/a	ok	n/a	ok	ok	ok
	I2C 1 own address register setup	1.5.4	ok	n/a	n/a	n/a	ok	ok	n/a
	I2C 1 interrupt enable register setup	1.5.5	ok	n/a	n/a	n/a	ok	ok	n/a
	I2C 1 Slave Address register setup (Master only)	1.5.6	ok	n/a	n/a	n/a	ok	ok	n/a
	Raw Rx buffer setup	1.5.7	ok	n/a	n/a	n/a	n/a	ok	ok

TEST RECORD: SYSTEM INITIALISATION		Software Checkup Matrix							
		Logic Step	Function Code	Error handling	Variables	Directives	Arguments	Logic	Opt.
OSFW_C6727_SPI_Init()		1.6	ok	n/a	n/a	n/a	n/a	ok	n/a
	SPI Default Chip Select Register (SPIDEF)	1.6.1	ok	n/a	n/a	n/a	ok	ok	n/a
	SPI Global Control Register 0 (SPIGCR0)	1.6.2	ok	n/a	n/a	n/a	ok	ok	n/a
	SPI Global Control Register 1 (SPIGCR1)	1.6.3	ok	n/a	ok	n/a	ok	ok	ok
	SPI Pin Control Register (SPIPC0)	1.6.4	ok	n/a	n/a	n/a	ok	ok	n/a
	SPI Shift register (SPIDAT1)	1.6.5	ok	n/a	ok	n/a	ok	ok	ok
	SPI Data Format Register 1 (SPIFMT1)	1.6.6	ok	n/a	ok	n/a	ok	ok	ok
	SPI Delay Register (SPIDELAY)	1.6.7	ok	n/a	ok	n/a	ok	ok	ok
	SPI Interrupt register 1 (SPIINT0)	1.6.8	ok	n/a	ok	n/a	ok	ok	ok
	SPI Interrupt Level Register (SPILVL)	1.6.9	ok	n/a	n/a	n/a	ok	ok	n/a
	Enable SPI communication	1.6.10	ok	n/a	n/a	n/a	ok	ok	n/a
	Enable SPI 1 DMA	1.6.11	ok	n/a	n/a	n/a	ok	ok	n/a
OSFW_C6727_Timer_Init()		1.7	ok	n/a	n/a	n/a	n/a	ok	n/a
	Disable the RTI counters	1.7.1	ok	n/a	n/a	n/a	ok	ok	n/a
	Clear RTI Up counter 0 and Free Running counter 0	1.7.2	ok	n/a	n/a	n/a	ok	ok	n/a
	Set the RTI compare Up counter 0 register	1.7.3	ok	n/a	ok	n/a	ok	ok	ok
	Set the RTI Update Compare 0 and Compare 0 registers	1.7.4	ok	n/a	n/a	n/a	ok	ok	n/a
	Select RTI Free Running Counter 0 as the Counter	1.7.5	ok	n/a	n/a	n/a	ok	ok	n/a
	Set RTI interrupt to Compare 0	1.7.6	ok	n/a	n/a	n/a	ok	ok	n/a
	Enable Counter 0	1.7.7	ok	n/a	n/a	n/a	ok	ok	n/a
OSFW_6727_IRQ_Init()		1.8	ok	n/a	n/a	n/a	n/a	ok	n/a
	Enable interrupts needed by OSFW	1.8.1	ok	n/a	n/a	n/a	n/a	ok	n/a

TEST RECORD: SYSTEM INITIALISATION		Software Checkup Matrix							
		Logic Step	Function Code	Error handling	Variables	Directives	Arguments	Logic	Opt.
OSFW_MemInit()		2.1	ok	n/a	n/a	n/a	ok	ok	ok
	Initialise OSFW's SDRAM memory	2.1.1	ok	n/a	ok	ok	ok	ok	ok
	Initialise OSFW's FPGA memory	2.1.2	untested	n/a	ok	ok	ok	ok	ok
	Memory allocation for the FPGA I/O data area in DSP	2.1.3	ok	n/a	n/a	n/a	ok	ok	n/a
	Memory allocation for dynamic memory allocation needs	2.1.4	ok	n/a	n/a	n/a	ok	ok	n/a
	Memory allocation: a 256 byte block	2.1.5	ok	n/a	n/a	ok	ok	ok	n/a
	Memory allocation: a 512 byte block	2.1.6	ok	n/a	n/a	ok	ok	ok	n/a
	Memory allocation: a 1 Kbyte block	2.1.7	ok	n/a	n/a	ok	ok	ok	n/a
	Memory allocation: a 2 Kbyte block	2.1.8	ok	n/a	n/a	ok	ok	ok	n/a
	Memory allocation: a 4 Kbyte block	2.1.9	ok	n/a	n/a	ok	ok	ok	n/a
	Memory allocation: a 8 Kbyte block	2.1.10	ok	n/a	n/a	ok	ok	ok	n/a
	Memory allocation: a 16 Kbyte block	2.1.11	ok	n/a	n/a	ok	ok	ok	n/a
	Memory allocation for task data blocks	2.1.12	ok	n/a	n/a	n/a	ok	ok	n/a
	Memory allocation for the error buffer	2.1.13	ok	n/a	n/a	n/a	ok	ok	n/a
	Memory allocation for the command buffer	2.1.14	ok	n/a	n/a	n/a	ok	ok	n/a
	Memory allocation for the data collection buffer	2.1.15	ok	n/a	n/a	ok	ok	ok	n/a
	Memory allocation for the ring-buffer objects space	2.1.16	ok	n/a	n/a	ok	ok	ok	n/a
	Memory allocation for the Tx buffer	2.1.17	ok	n/a	n/a	n/a	ok	ok	n/a
	Memory allocation for the Rx buffer	2.1.18	ok	n/a	n/a	n/a	ok	ok	n/a
	Memory allocation for miscellaneous needs	2.1.19	ok	n/a	n/a	ok	ok	ok	n/a
OSFW_ErrorQueueInit()		2.2	ok	n/a	n/a	n/a	ok	ok	n/a
OSFW_CommBufQueueInit()		2.3	ok	n/a	n/a	n/a	ok	ok	n/a
OSFW_CommandQueueInit()		2.4	ok	n/a	ok	ok	ok	ok	ok

TEST RECORD: SYSTEM INITIALISATION		Software Checkup Matrix							
		Logic Step	Function Code	Error handling	Variables	Directives	Arguments	Logic	Opt.
OSFW_FPGA_IOMemInit()		2.5	ok	n/a	n/a	n/a	n/a	ok	ok
	Initialise FPGA memory I/O pointers	2.5.1	ok	n/a	n/a	n/a	ok	ok	ok
	Reserve SDRAM memory for I/O object	2.5.2	ok	ok	n/a	ok	ok	ok	n/a
	Set addresses of the global I/O variables	2.5.3	ok	n/a	n/a	ok	n/a	ok	ok
OSFW_EventFlagInit()		2.6	ok	ok	n/a	n/a	n/a	ok	n/a
Create OSFW tasks		2.7	ok	ok	n/a	n/a	ok	ok	n/a
Create OSFW semaphores		2.8	ok	n/a	n/a	n/a	ok	ok	n/a
Initialise the Watchdog		2.9	ok	n/a	n/a	ok	ok	ok	ok

The OSFW Testing Record: Functions and System APIs

TEST RECORD: SYSTEM FUNCTIONS		Software Checkup Matrix						
		Function Code	Error handling	Variables	Directives	Arguments	Logic	Opt.
OSFW tests	AMBContr()	untested	untested	untested	untested	untested	untested	untested
	Memory API Tests	ok	n/a	ok	ok	ok	ok	n/a
	Error API Tests	ok	n/a	ok	ok	ok	ok	n/a
	Communication API Tests	ok	n/a	ok	ok	ok	ok	n/a
	Command API Tests	ok	n/a	ok	ok	ok	ok	n/a
	Buffer API Tests	ok	n/a	ok	ok	ok	ok	n/a
	ISO/IEC Substitutes	ok	n/a	ok	ok	ok	ok	n/a
	Interrupt Service Routine Tests	ok	n/a	n/a	ok	n/a	ok	n/a
	Main() Function	ok	ok	ok	n/a	n/a	ok	ok
RTOS Port	DSP_C6x_Vectors.asm	untested	n/a	n/a	n/a	n/a	ok	n/a
	os_cpu_a.asm	ok	n/a	ok	n/a	n/a	ok	ok
	os_cpu_c.c	ok	n/a	ok	ok	ok	ok	ok
Interrupt Service Routines	EMIF_NMI_ISR()	untested	n/a	n/a	untested	n/a	untested	untested
	I2Cx_SPIx_ISR()	ok	ok	ok	n/a	n/a	ok	ok
	dMAX_TCINT_ISR()	ok	n/a	ok	n/a	n/a	ok	ok
RTOS extensions	OSEnterCriticalHook	ok	n/a	n/a	n/a	n/a	ok	ok
	OSExitCriticalHook	ok	n/a	n/a	n/a	n/a	ok	ok
C99 Substitutes	OSFW_printf()	ok	ok	ok	ok	ok	ok	n/a
	OSFW_strtok_r()	ok	ok	ok	ok	ok	ok	n/a

TEST RECORD: SYSTEM FUNCTIONS		Software Checkup Matrix						
		Function Code	Error handling	Variables	Directives	Arguments	Logic	Opt.
Memory API	OSFW_MemCheck()	ok	ok	ok	ok	ok	ok	ok(rev)
	OSFW_MemCopy()	ok	n/a	n/a	n/a	ok	ok	ok
	OSFW_MemClear()	ok	n/a	n/a	n/a	ok	ok	ok
	OSFW_TestBit()	ok	n/a	ok	n/a	ok	ok	ok
	OSFW_SetWordRegisterBit()	ok	n/a	ok	n/a	ok	ok	ok
	OSFW_ResetWordRegisterBit()	ok	n/a	ok	n/a	ok	ok	ok
	OSFW_SetByteRegisterBit()	ok	n/a	ok	n/a	ok	ok	ok
	OSFW_ResetByteRegisterBit()	ok	n/a	ok	n/a	ok	ok	ok
	OSFW_SetWordRegisterMask()	ok	n/a	n/a	n/a	ok	ok	ok
	OSFW_ResetWordRegisterMask()	ok	n/a	n/a	n/a	ok	ok	ok
	OSFW_SetRegister()	ok	n/a	ok	n/a	ok	ok	ok
	OSFW_ClearInterrupt()	ok	n/a	n/a	n/a	ok	ok	ok
	OSFW_EnableInterrupt()	ok	n/a	n/a	n/a	ok	ok	ok
	OSFW_DisableInterrupt()	ok	n/a	n/a	n/a	ok	ok	ok
	OSFW_IntToFloat()	ok	n/a	ok	n/a	ok	ok	ok
	OSFW_LongToFloat()	ok	n/a	ok	n/a	ok	ok	ok
	OSFW_IntToDouble()	ok	n/a	ok	n/a	ok	ok	ok
	OSFW_LongToDouble()	ok	n/a	ok	n/a	ok	ok	ok
	OSFW_FloatToInt()	ok	n/a	ok	n/a	ok	ok	ok
	OSFW_DoubleToInt()	ok	n/a	ok	n/a	ok	ok	ok
	OSFW_FloatToLong()	ok	n/a	ok	n/a	ok	ok	ok
	OSFW_DoubleToLong()	ok	n/a	ok	n/a	ok	ok	ok
	OSFW_NopDelay()	ok	n/a	ok	n/a	ok	ok	ok
	OSFW_C6727_Watchdog()	ok	n/a	n/a	n/a	ok	ok	ok

TEST RECORD: SYSTEM FUNCTIONS		Software Checkup Matrix						
		Function Code	Error handling	Variables	Directives	Arguments	Logic	Opt.
Communication API	OSFW_CommBufQueueInit()	ok	n/a	n/a	n/a	ok	ok	n/a
	OSFW_EventFlagInit()	ok	ok	n/a	n/a	n/a	ok	n/a
	OSFW_CommDataSend()	ok	ok	ok	ok	ok	ok	n/a
	OSFW_CommDataParser()	ok	ok	ok	ok	ok	ok	ok
	OSFW_CRC()	requires development	n/a	ok	n/a	ok	ok	ok
	OSFW_RxHwDriver()	ok	n/a	ok	n/a	ok	ok	ok
	OSFW_TxHwDriver()	ok	n/a	ok	n/a	ok	ok	n/a
	OSFW_TaskCommTX()	ok	ok	ok	ok	ok	ok	n/a
	OSFW_TaskCommRX()	ok	ok	ok	ok	ok	ok	n/a
	OSFW_FPGA_IOMemInit()	ok	n/a	n/a	n/a	n/a	ok	ok
	OSFW_FPGA_IODriver()	ok	n/a	ok	n/a	ok	ok	ok(rev)
Command API	OSFW_CommandQueueInit()	ok	n/a	ok	n/a	ok	ok	ok
	OSFW_DataObjCreate()	ok	ok	ok	n/a	ok	ok	ok
	OSFW_CommandInterpreter()	ok	n/a	ok	n/a	ok	ok	ok
	OSFW_TaskCommandHandler()	ok	ok	ok	ok	ok	ok	ok
Buffer API	OSFW_RingBufferInit()	ok	ok	ok	n/a	ok	ok	ok
	OSFW_RingBufferCheck()	ok	n/a	ok	n/a	ok	ok	ok
	OSFW_RingBufferAdd()	ok	ok	ok	ok	ok	ok	n/a
	OSFW_RingBufferClear()	ok	ok	ok	ok	ok	ok	n/a
	OSFW_RingBufferCreate()	ok	ok	ok	ok	ok	ok	ok
	OSFW_RingBufferDelete()	ok	ok	ok	ok	ok	ok	n/a

TEST RECORD: SYSTEM FUNCTIONS		Software Checkup Matrix						
		Function Code	Error handling	Variables	Directives	Arguments	Logic	Opt.
Error API	OSFW_ErrorQueueInit()	ok	n/a	n/a	n/a	ok	ok	n/a
	OSFW_ErrorMsgGen()	ok	ok	n/a	ok	ok	ok	n/a
	OSFW_ErrorReport()	ok	ok	ok	ok	ok	ok	n/a
	OSFW_TaskErrHandler()	ok	ok	ok	ok	ok	ok	n/a
	OSFW_TaskSPIErrHandler()	requires development	ok	ok	ok	ok	ok	ok
	OSFW_TaskWatchdog()	ok	n/a	n/a	n/a	ok	ok	ok