AN EVALUATION OF ASPECT-ORIENTED PROGRAMMING FOR EMBEDDED REAL-TIME SYSTEMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

YUSUF BORA KARTAL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

MAY 2007

Approval of the Graduate School of (Name of the Graduate School)

Prof. Dr. Canan ÖZGEN
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. İsmet ERKMEN
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Dr. Şenan Ece SCHMIDT
Supervisor

Examining Committee Members

Prof. Dr. Semih BİLGEN (METU, EE)

Dr. Şenan Ece SCHMIDT (METU, EE)

Prof. Dr. Hasan GÜRAN (METU, EE)

Asst. Prof. Dr. Cüneyt BAZLAMAÇCI (METU, EE)

Hakkı Özgür GÖREN (ASELSAN)

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Yusuf Bora KARTAL

Signature :

ABSTRACT

AN EVALUATION OF ASPECT ORIENTED PROGRAMMING FOR EMBEDDED REAL-TIME SYSTEMS

KARTAL, Yusuf Bora

M.S., Department of Electrical and Electronics Engineering

Supervisor : Dr. Şenan Ece SCHMIDT

May 2007, 81 pages

Crosscutting concerns are the issues in software that cannot be modularized within a software module. In this thesis work, a detailed evaluation of the use of Aspect Oriented Programming for the implementation of crosscutting concerns in embedded real-time systems is presented. The pilot Audio Switch project implementations are first evaluated in terms of software quality attributes. Then a detailed analysis of the two implementations, according to embedded real-time performance metrics has been carried out. Evaluation results show the benefits of Aspect Oriented Programming in embedded real-time systems.

Keywords: Aspect Oriented Programming, Crosscutting Concerns, Embedded Real-Time Systems

ÖZ

İLGİYE ODAKLI PROGRAMLAMANIN GERÇEK ZAMANLI GÖMÜLÜ SİSTEMLER ÜZERİNDE BİR DEĞERLENDİRMESİ

KARTAL, Yusuf Bora

Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü

Tez Yöneticisi : Dr. Şenan Ece SCHMIDT

Mayıs 2007, 81 sayfa

Enine-kesen ilgiler tek bir yazılım parçasının içinde gerçeklenemeyip birden fazla parçaya yayılmış olan işlerdir. Bu tez çalışmasında, İlgiye Odaklı Programlama'nın gerçek-zamanlı gömülü sistemlerdeki enine-kesen ilgilerin gerçeklenmesindeki kullanımı değerlendirilmektedir. Örnek Ses Anahtarı projesinin gerçeklemeleri, öncelikle yazılım kalite özniteliklerine göre değerlendirilmiştir. Daha sonra, gerçeklemelerin, gerçek-zamanlı gömülü sistem performans metriklerine göre değerlendirmesi yapılmıştır. Değerlendirme sonuçları, İlgiye Odaklı Programlamanın gerçek-zamanlı gömülü sistemlerdeki kullanımının getirdiği faydaları ortaya koymaktadır.

Anahtar Kelimeler: İlgiye Odaklı Programlama, Enine-Kesen İlgiler, Gerçek-Zamanlı Gömülü Sistemler

To My Parents and To My Dear

ACKNOWLEDGMENTS

I would like to thank Dr. Şenan Ece SCHMIDT for her valuable supervision, support and tolerance throughout the development and improvement of this thesis.

I am grateful to Hakkı Özgür GÖREN for his support throughout the development and the improvement of this thesis. I am also grateful to Aselsan Electronics Industries Inc. for the resources and facilities that I use throughout thesis.

Thanks a lot to all my friends for their great encouragement and their valuable help to accomplish this work.

Finally, I would like to thank to my parents for bringing up and trusting in me, and to my dear, whom I love in deep, for just being there when I need.

TABLE OF CONTENTS

ABSTRACT iv
ÖZ v
ACKNOWLEDGMENTS vii
TABLE OF CONTENTS viii
LIST OF TABLES xi
LIST OF FIGURES xii
LIST OF ABBREVIATIONS xiv
INTRODUCTION 1
CHAPTER II 5
BACKGROUND 5
2.1 Real-Time Systems 5
2.1.1 Characteristics of Real-Time Systems 7
2.1.2 Hard Real-Time Systems 7
2.1.3 Soft Real-Time Systems 8
2.2 Embedded Systems 8
2.3 Object-Oriented Design 11
2.3.1 Object-Oriented Design Principles 14
2.3.1.1 Single Responsibility Principle (SRP) 14
2.3.1.2 Open Closed Principle (OCP) 14
2.3.1.3 Interface Segregation Principle (ISP) 15
2.3.1.4 Liskov Substitution Principle (LSP) 15
2.3.1.5 Dependency Inversion Principle (DIP) 16
2.3.1.6 Common Closure Principle (CCP) 16
2.3.1.7 Stable Dependencies Principle (SDP) 17
2.3.1.8 Stable Abstractions Principle (SAP) 18
2.4 Crosscutting Concerns 19
2.5 Aspect Oriented Programming 21
2.5.1 How Aspect Oriented Programming Works 23

2.5.2 An Aspect Language: AspectC++ 28
IMPLEMENTATION 32
3.1 Case Study 32
3.1.1 Motorola MVME 5100 Board 33
3.1.2 Real-Time Operating System VxWorks 35
3.2 Project Description 36
3.2.1 A/D Converter Block 37
3.2.2 Data Processing Block 37
3.2.3 D/A Converter Block 38
3.3 Implemented Non-Functional Concerns 41
3.3.1 Logging Concern 41
3.3.2 Error Checking Concern: 43
3.3.3 Range Checking Concern: 44
3.3.4 Real-Time Property Concern: 46
EVALUATION 48
4.1 Software Quality 49
4.1.1 Chidamber and Kemerer Metrics Suite 50
4.1.1.1 Weighted Methods Per Class (WMC) 50
4.1.1.2 Coupling Between Objects (CBO) 52
4.1.1.3 Response For A Class (RFC) 53
4.1.1.4 Lack Of Cohesion In Methods (LCOM) 55
4.1.1.5 Depth of Inheritance Tree (DIT) 56
4.1.1.6 Number Of Children (NOC) 57
4.1.2 Software Quality Results Summary of Audio Switch Project 58
4.2 Embedded Real-Time System Performance 59
4.2.1 Memory Usage 60
4.2.2 CPU Usage 61
4.2.3 Run-Time 63
4.2.3.1 A/D Converter Block Run-Time Results 64
4.2.3.2 D/A Converter Block Run-Time Results 66
4.2.3.3 Data Processing Block Run-Time Results 67
4.3 Summary of Embedded Real-Time System Performance 69
CONCLUSION 72

REFERENCES 74
APPENDIX A 77
AspectC++ Language Quick Reference 77
APPENDIX B 79
Motorola MVME 5100 Specifications... 79

LIST OF TABLES

Table 4.1 Mapping of C&K Metrics on Software Quality Attributes 58
Table 4.2 Total Effects of AOP on Software Quality Metrics 58
Table 4.3 Effects of AOP Usage on Software Quality Attributes 59
Table 4.4 Effects of AOP on Embedded Real-Time Metrics 70
Table 4.5 Embedded Real-Time Metrics Metrics Summary 70

LIST OF FIGURES

Figure 2.1 Real-Time System Mechanisms [7] 6
Figure 2.2 Components of an Embedded System 9
Figure 2.3 A/D Converter Object.. 12
Figure 2.4 A System Made up of Interacting Objects [23]............................ 13
Figure 2.5 Sample Dependency Relation 17
Figure 2.6 Sample application of SAP in UML Notation.............................. 18
Figure 2.7 Sample Orientation of Modules in a Software Project [11].......... 20
Figure 2.8 Use of Aspect Oriented Programming [24] 23
Figure 2.9 Aspect Weaver 24
Figure 2.10 Class Diagram of Object-Oriented Implementation of Logging Concern in Audio Switch Project.. 25
Figure 2.11 Object-Oriented Implementation of Logging Concern in Audio Switch Project 25
Figure 2.12 Logical Settlement of Aspect-Oriented Implementation of Logging Concern in Audio Switch Project.. 26
Figure 2.13 Aspect-Oriented Implementation of Logging Concern in Audio Switch Project 26
Figure 2.14 Aspect-Oriented Implementation of Logging Concern 27
Figure 3.1 Thumbnail of MVME 5100 [17] 34
Figure 3.2 Software modules of Audio Switch Project 36
Figure 3.3 Summary of Audio Switch Project Operation 40
Figure 3.4 Object-Oriented Implementation of Logging Concern................. 41
Figure 3.5 Aspect-Oriented Implementation of Logging Concern 42
Figure 3.6 Object-Oriented Implementation of Error Checking Concern...... 43
Figure 3.7 Aspect-Oriented Implementation of Error Checking Concern 44
Figure 3.8 Object-Oriented Implementation of Range Checking Concern ... 45
Figure 3.9 Aspect-Oriented Implementation of Range Checking Concern... 45

Figure 3.10 Object-Oriented Implementation of Real-Time Property Concern 46
Figure 3.11 Aspect-Oriented Implementation of Real-Time Property Concern 47
Figure 4.1 WMC Metric Results 51
Figure 4.2 CBO Metric Results 53
Figure 4.3 RFC Metric Results.. 54
Figure 4.4 LCOM Metric Results.. 56
Figure 4.5 Dynamic Memory Usage Results.. 61
Figure 4.6 CPU Usage Results 63
Figure 4.7 Average Run-Time Measurement Results of A/D Converter Block 65
Figure 4.8 Worst Case Run-Time Measurement Results of A/D Converter Block 65
Figure 4.9 Average Run-Time Measurement Results of D/A Converter Block 66
Figure 4.10 Worst Case Run-Time Measurement Results of D/A Converter Block 67
Figure 4.11 Average Run-Time Measurement Results of Data Processing Block 68
Figure 4.12 Worst Case Run-Time Measurement Results of Data Processing Block 68

LIST OF ABBREVIATIONS

AOP : Aspect Oriented Programming
A/D : Analog to Digital
C&K : Chidamber and Kemerer
DAC : Digital to Analog Converter
D/A : Digital to Analog
METU : Middle East Technical University
OOP : Object Oriented Programming
RTOS : Real-Time Operating System
SOC : Separation of Concerns
UML : Unified Modeling Language

CHAPTER I

INTRODUCTION

Advances in programming languages have aimed to improve the software developers' ability to build up more modular code. Especially in desktop computing, software quality becomes one of the major components of the system. Modularity in software implies reusability, maintainability and testability of the system.

Nowadays, Object Oriented Programming (OOP) is the dominant programming paradigm where the real-life objects are mapped to software objects as an abstraction. OOP is a way to develop modular software; however it still has some limitations in the field of Separation of Concerns (SOC), which refers to the identification and encapsulation of different concerns in different software blocks.

OOP is good at separating the functional concerns of the software, which defines the core functionality of the system. However, system software is not only composed of functional concerns. Besides the functional concerns, there are some non-functional concerns like logging, error handling, which are especially used in the software development life cycle. These concerns, when implemented by the OOP techniques, have a crosscutting behavior over the functional software blocks. Their implementations are spread over many software modules. Because of this crosscutting behavior, these concerns are named as Crosscutting Concerns.

Crosscutting concerns hinder the modularity of the system software and degrade the software quality of the system. To avoid this degradation, SOC is needed for the modularization of crosscutting concerns.

Aspect-Oriented Programming (AOP), which is built on the existing OOP techniques, is the latest advancement in programming techniques in the pursuit of SOC. AOP has proposed new concepts for the programmers to easily modularize and control the crosscutting functionality of the software. A large number of studies are carried out to show the use of AOP in non-real-time desktop computing systems. However application of AOP in the field of embedded real-time systems is not fully studied yet.

Embedded real-time systems differ significantly from the desktop computing systems. They have special requirements and resource constraints, which drive the software development process. The software developed for an embedded real-time system should be predictable, and should be able to work with the least powerful computers that can meet the functional and performance requirements of the system. [1]

Traditionally, using the procedural programming languages like C is used in developing embedded real-time system software. Moreover the low level Assembly language is used to develop most of the embedded applications. However none of these languages have the simplicity of OOP languages. OOP is not yet fully integrated to the embedded real-time software development process because of its overhead.

The main reason for the performance overhead of OOP is the message passing and context switching issues. Especially the overhead of the crosscutting concerns is not tolerable.

In this thesis, the evaluation of Aspect Oriented Programming is made for the embedded real-time systems on the Audio Switch project. A comparison between AOP and OOP is made according to both software quality attributes and embedded real-time performance metrics.

Audio Switch project is a software implementation of an audio matrix. There are forty input channels, each of which can be switched to sixteen different audio outputs separately. The switch can be controlled via a graphical user interface. The user can increase or decrease the signal levels of each input channel. Moreover the user can add a volume offset or completely mute any

input channels. Each input channel can be switched to one or more output channels. Besides these, the user has the ability to multiplex several input channels to one or more output channels.

The aim of the research is to determine if AOP is techniques provide better separation of crosscutting concerns in the field of embedded real-time systems. Moreover, it is tried to figure out if AOP gives better performance results in terms of embedded real-time performance metrics.

In this work, an embedded real-time application is developed to compare the performance of object-oriented and aspect-oriented programming techniques in the implementation of non-functional crosscutting concerns.

As stated in [21, 25] Chidamber and Kemerer (C&K) Metric Suite provides the most comprehensive and best-validated set of measures. C&K metrics suite was generated to fulfill the need for an evaluation metrics suite for Object-Oriented Design methodology. These metrics give numerical results to measure the four software attributes of the system. These metrics are proposed by Shyam R. Chidamber and Chris F. Kemerer in [20] and widely adopted for evaluating the quality of object-oriented system design.

The two implementations are first compared according the Chidamber and Kemerer Metrics Suite to see the difference of the implementations in terms of software quality attributes. Then, another comparison is carried out to see the differences of the two implementations in terms of embedded real-time performance metrics. Percent CPU usage, percent memory usage and run-time differences are examined in this context.

The remainder of this thesis organized as follows: In Chapter II background information about embedded real-time systems, Object Oriented Programming and Aspect Oriented Programming is given. Moreover the phenomenon of Separation of Concerns and Crosscutting Concerns are also discussed in this chapter.

In Chapter III the pilot Audio Switch project is discussed. The operating environment and the implementation of the project are described in detail. In

this chapter the implementation is divided into three sub-blocks. Each of these sub-blocks and their relations are described in detail.

In Chapter IV the evaluation results of the Audio Switch project are given. The results given in this chapter are divided into two sub-groups, according to the evaluation procedures. In the first group the evaluation results of the software quality metrics are given. In the second group, the evaluation results of the embedded real-time performance metrics are explored in detail.

Finally in Chapter V the evaluation results and the implementation itself are summarized. Moreover, the advantages and disadvantages of using AOP in the implementation of crosscutting concerns in embedded real-time systems are presented in this chapter. In addition to these, some possible improvements that can be gathered using AOP in embedded real-time systems are discussed in this chapter.

CHAPTER II

BACKGROUND

This chapter describes the embedded systems, real-time systems; object-oriented programming and the aspect-oriented programming paradigms. In particular, object-oriented and aspect-oriented programming paradigms will be addressed.

2.1 Real-Time Systems

Systems in which, the production time of the output is as significant as the output itself, are defined as real-time systems [1]. In real-time systems, each input corresponds to an action in the real domain, so the output of the system should relate with the reaction in the real domain. The lag between the input and output times of a real-time system should be acceptably small.

Because of their real domain relations, the correctness of real-time systems is not only related with the correctness of their outputs but also the production time of the outputs. The correct function at the correct time is the key concept for real-time systems. A real-time system, which should complete a job in 10 microseconds, should complete that job within that time interval, otherwise the system fails. Unpredictable delays in real-time systems are not acceptable.

Real-time systems can be found in many different areas ranging from control systems to multimedia video conferencing.

Most real-time systems operate in mission critical environment in which events must be processed in a strict order within bounded delays. Hence the

real-time systems must provide a set of real-time mechanisms or services. Five standard mechanisms are: Task Management, Interprocess Communication, Dynamic Memory Allocation, Device I/O Management and Timers. Figure 2.1 shows the interaction between these mechanisms in a real-time system:

Figure 2.1 Real-Time System Mechanisms [7]

The first mechanism, task management, is the priority-based scheduling and management of processes. The second is inter-process communication, which allows processes to pass messages between each other in a reliable and timely manner. Dynamic memory allocation provides processes with dynamic pools of memory that can be shared by multiple processes for the fast sharing of data. Device I/O management controls all devices in a real-time fashion. Timer services offered by real-time systems include task delay and time-outs, and they are used to enforce the bounded timing requirements. [7]

In general the real-time systems are either event triggered based on interrupts or time-triggered based on deadlines. [2]

2.1.1 Characteristics of Real-Time Systems

According to [3,4,5] the real-time systems have some common distinguishing characteristics. These characteristics are as follows:

- Real-time systems are **predictable**. The time needed for a job is within predictable limits.
- Real-time systems are **responsive**. They interact with their surroundings via some peripherals.
- They are **robust**. They give correct results under unpredictable and even erroneous conditions.
- They are usually **embedded** within larger systems such that their behavior is indistinguishable from that of the entire system.
- They are often **distributed**.
- They do their job by executing multiple tasks **concurrently** regarding their priorities.

Real-time systems have deadlines to complete their jobs. They are divided into two distinct groups according to the strictness of the timing limits: hard real-time and soft real-time systems.

2.1.2 Hard Real-Time Systems

A hard real-time system is a system in which the distribution of its met and missed deadlines during a window of time w is precisely bounded. [6] The timing constraints of hard real-time systems should be met precisely.

Even a small delay in such systems is unacceptable. In hard-real time systems delay means failure in the system operation. The automatic breaking system in the automobiles is a typical example for hard real-time systems. A

delay in the system operation means failure and can cause a crash. Missing the deadlines may result in catastrophic consequences or loss of system performance.

2.1.3 Soft Real-Time Systems

Soft real-time systems are the systems in which, missing some of the deadlines is occasionally acceptable. However it is still important to fulfill the timing requirements. In other words, the position of the missing dead lines is important. Consecutive misses may cause system failure.

In these systems delay does not always mean failure for the system. Real-time multimedia broadcasting is a typical example for soft real-time systems. Small delays in the file transfer can be acceptable if the user does not recognize it. However it is still important not to exceed the timing limits for data quality. [7]

To summarize a real-time system is a system, which should provide an appropriate response within a certain time bound

2.2 Embedded Systems

An embedded system is a physical system that employs computer control for a specific purpose. Unlike a general purpose computing system, an embedded system does one or a few predefined tasks. Embedded systems do not provide standard computing services and they usually form a part of a larger system. [1]

Embedded systems usually have limited user interface. They usually operate for long time periods. The computer in an embedded system is strictly related to its operating environment (peripherals). A computerized dishwasher is a

typical example for an embedded system where the main system provides a non-computing function with the help of an embedded computer.

A typical embedded system has a central processing unit (CPU), a main memory unit (MMU) and its peripherals such as device drivers, converters and interfaces. The CPU is responsible for computing the algorithms running for the system operation. The input and output data are kept in the main memory unit.

The peripherals in an embedded system are usually specialized device drivers, which give control service for the use of specific hardware. The system communicates with the outside world via these peripherals. Figure 2.2 shows the typical components of an embedded system.

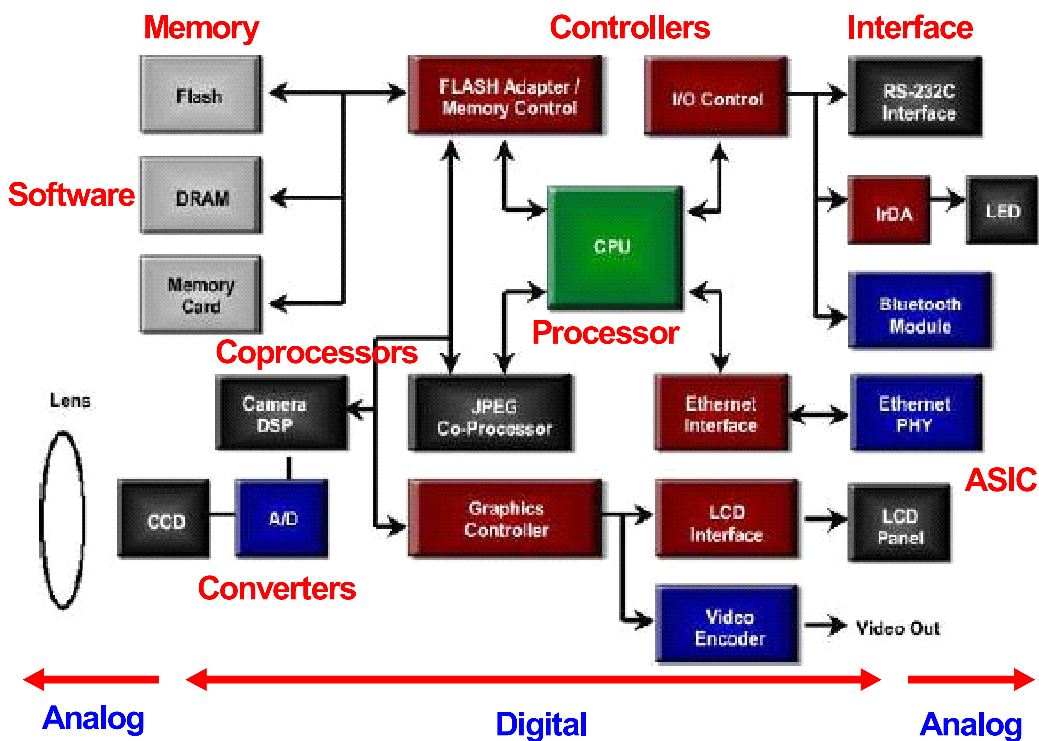

Figure 2.2 Components of an Embedded System

An embedded system is not always a separate block. It is usually built in the device that it is controlling. “Embedded systems are usually constructed with the least powerful computers that can meet the functional and performance requirements.”[1]

Embedded systems can be seen in a wide range of application areas ranging from medical applications (heart monitors) to military applications (weapon control systems).

Although embedded systems can be basic units, which are responsible for very specific jobs, they can also be very complex systems doing several jobs at the same time. Embedded systems have some resource constraints related to their work and operating environment.

In real-life, most of the embedded systems have real-time specifications. These kind of embedded systems are called Embedded Real-Time Systems.

The real-time embedded systems design becomes more and more complicated, with the increasing demands from the embedded systems and from the nature of being real-time. [8] They have to control different hardware. Besides this, they have to have higher performance and reliability. These requirements change the development style of embedded systems.

One of the main characteristics of the embedded systems is their context dependency. The term context dependency here implies the environment in which the system is operating. Because of being context dependent the system should be able to change its behavior with respect to the environmental changes.

Context-dependent systems have three important conceptual parts: One part consists of the sensors, which provides the communication of the system with its environment. The second part is made up of the logic to decide contexts based on the data gathered from those sensors. And the last part consists of the internal processing that is triggered by the determined contexts. [8]

As the embedded systems become more and more complicated, the software controlling those systems are becoming more and more

complicated though. The increasing complexity in the software requires a new design approach. Nowadays Object Oriented Design is the most popular and dominant design methodology for embedded systems software.

2.3 Object-Oriented Design

Object-oriented design is nowadays the most popular and dominant development paradigm in the software development life cycle. Before going through the details and main principles of object-oriented design, it is worth to give the meaning of the term “object” and show the mapping between the real domain problems and software objects.

The term “object” is now being widely used in the design process. An object is an encapsulation of the information related to a specific task. It is an entity that keeps state information and has defined operations controlling its functionality. The operations of an object provide an interface to the other objects. The object functionality is controlled by both its state and those predefined operations. Object functionality, in other words, its response to outside effects is determined by the called operation and its current state.

Objects keep their state information in their attributes. Hence it can be said that the reaction of an object to an outside action is decided by looking at the attributes and methods of that object. Here the term “method” is used to refer to the implementation of an object operation. Figure 2.3 shows a software object with its attributes and operations.

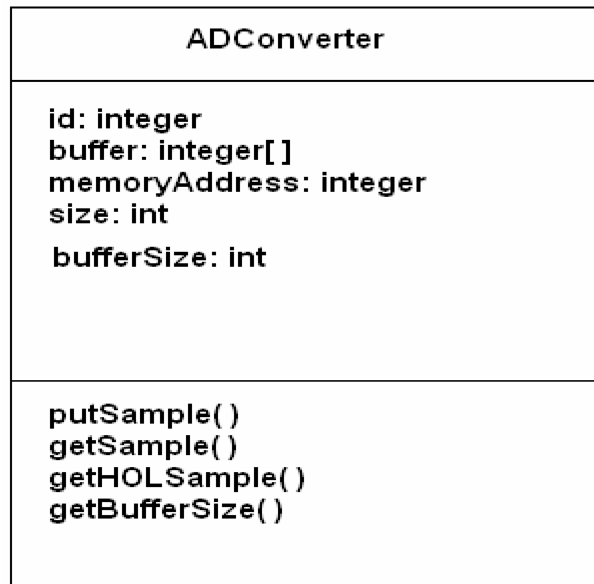

Figure 2.3 A/D Converter Object

The objects in the software are the mappings of the real-domain objects. In Figure 2.3 the A/D converter object is the software controller of the analog to digital converter in the real-life. Its job is, coordinating the sample flow in the A/D converter memory map.

As in this case the software objects in the object oriented design approach are the mappings of the real objects in the real-domain problems. Hence it is easy to construct software for a real-life problem by using object-oriented design methodology.

In a system software, each object keeps state information related to a specific job. The entire system operation is performed by the interaction between the objects. The interactions between the software objects are performed via message passing. The common region between the software objects is lowered in the object oriented design methodology. Figure 2.4 illustrates the message passing between the objects in an object-oriented design.

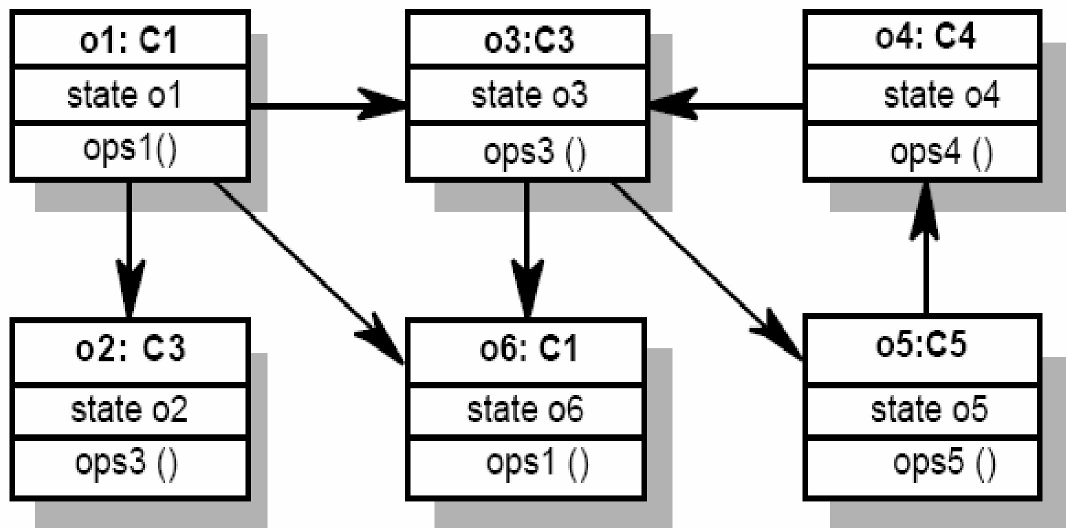

Figure 2.4 A System Made up of Interacting Objects [23]

In some distributed systems the message passing between the software objects is performed by text messages. The receiving entity parses the message and does the requested job in that message. But if the software objects exist in the same program the message passing is performed via method call or event passing.

There are several principles that object-oriented design relies on. The main principles of object-oriented design are the principle of “high cohesion” and “low coupling”.

The term “cohesion” in software development, is a measure of how strongly the total lines of code in an object’s implementation work together to do a specific job. In software development, high cohesion is preferable, high cohesive objects, in other words objects dealing with a specific job are more adaptable to the environmental changes. Moreover high cohesion implies the reusability and maintainability of the software object.

The term “coupling” in software development is a measure of the degree to which each module in the software relies on each of the other modules in the project. Low coupling is more preferable in object-oriented software design, because loosely coupled objects are easy to manage. In object-oriented

design a small change in an object may cause unmanageable changes in the modules, to which it is related.

Thus, high cohesion and low coupling are the key aspects of an object oriented software design.

2.3.1 Object-Oriented Design Principles

Martin et. al. [9] lists the eight principles of object-oriented module design that provide high cohesion and low coupling (dependency). Next, these eight principles are described.

2.3.1.1 Single Responsibility Principle (SRP)

This principle imposes that a software object in an object-oriented design should deal with only one specific job. If a class is related to more than one job, in other words, if an object has more than one responsibility, any change in one of those responsibilities may cause that object to change. This change may not be so straightforward regarding the other responsibilities that the object should deal with. Hence, it is difficult to modify, change or reuse that object. This makes the object to be unchangeable and rigid. Thinking of the starting point of object-oriented design, this is not the point that it desires to reach.

In fact the Single Responsibility Principle is the object-oriented design solution to the classic "Separation of Concerns" problem, which is the problem of having more than one concern within one software module.

2.3.1.2 Open Closed Principle (OCP)

A change in an object may cause a cascade of changes in the other objects that are dependent to the changing object. This causes a fragile formation in the object-oriented design. This situation is another form of rigidity.

The Open-Closed Principle is a design strategy to prevent the fragile formation of the software design. According to this principle a software entity should be closed to modifications but still open to extension via sub-classing or composition. Hence, by preventing changes in the original entity OCP tries to prevent the brittle software formation.

There are still some cases that modification in the original module is the only way to satisfy the software requirements. No extensions via sub-classing or composition are available for those cases.

2.3.1.3 Interface Segregation Principle (ISP)

If an interface in system software gives service to more than one object a small change in that interface affects the clients. If a method in the interface is changed, that forces unwanted changes in the other objects that are not using that method. This is because all the objects are coupled to the same interface.

Interface Segregation Principle aims to solve this problem. It states that an object should only depend on the narrowest interface that satisfies the object's requirements. If the interfaces in a system software design are separated according to this principle the changes in those interfaces are localized.

However there can be still some requirements that make two different objects coupled to the same interface.

2.3.1.4 Liskov Substitution Principle (LSP)

Liskov Substitution Principle states that, subtypes should be substitutable for their base types. In other words the behavior of the derived classes should not alter the behavior of their base classes in the ways that alter the behavior of the objects that are dependent to the base class.

2.3.1.5 Dependency Inversion Principle (DIP)

In a system software if an object is dependent on another object directly or indirectly in more than one ways that causes a fragile design. If a class A is dependent on a class B that is dependent on another class C. And if class A is also dependent on class C, that situation causes a fragile system formation, because both objects depend on unnecessary details.

The Dependency Inversion Principle aims to solve these problems by stating the following simple rules:

- “High level modules should not depend on low level modules”
- “High level and low level modules should depend on abstractions”
- “Abstractions should not depend on details, details should depend on abstractions.” [10]

Applications of Dependency Inversion Principle can be seen in layered architectures where higher layers are dependent on lower layers over some interface classes.

2.3.1.6 Common Closure Principle (CCP)

Common Closure Principle is in fact the analog of Single Responsibility Principle. It is just the package-applied version of SRP.

As in SRP Common Closure Principle aims to localize the change within packages by designing the packages such that they are dealing with single specific job.

Common Closure Principle states that, the software package with all of its components should be closed together against the same kinds of changes. One change made to a package should affect all the classes in that package and should not affect any other classes outside that package.

This principle aims to make the software more cohesive. So it becomes easy to maintain and reuse. It is useful when thinking of the functional

requirements of a system, but it is simply useless when considering the nonfunctional crosscutting concerns in the system software.

2.3.1.7 Stable Dependencies Principle (SDP)

Changes in the objects, forces their dependent objects to change. This changes become more difficult to manage with the increasing dependency relationships.

Stable Dependencies Principle aims to solve this problem by giving a stable dependency formation in the software design.

It states that the dependency between two objects should be from less stable to more stable object. A less stable object should be dependent to a more stable object, because less stable objects are open to changes and this changes have effects on the object's dependents.

So the dependency relation should be arranged such that the object that has the lowest probability of change should be at the bottom layer.

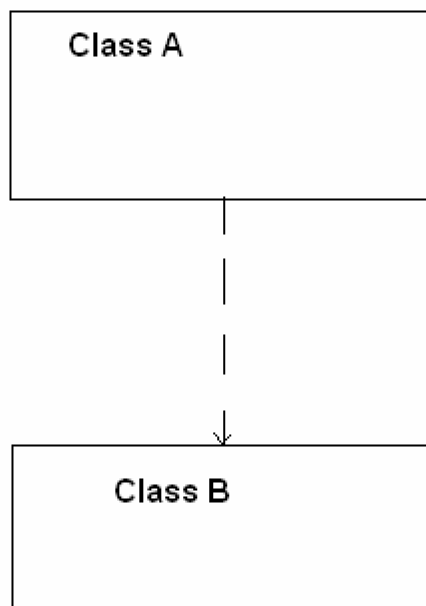

Figure 2.5 Sample Dependency Relation

In the above figure Class B should be more stable than class A, regarding the Stable Dependencies Principle.

2.3.1.8 Stable Abstractions Principle (SAP)

Stable Dependencies Principle guides software to stability. What can we do if we need flexibility in our design? The answer is Open Closed Principle. The system objects should be designed to be closed to modification and open to extensions.

Stable Abstractions Principle combines these two principles. It states that one can achieve stability and flexibility at the same time by using stable abstractions. Stability is achieved by putting the stable abstractions in different packages from the less stable implementations. Figure 2.6 illustrates this principle.

In this figure, ClassA takes service from ClassB via its interface class IClassB that is more stable than the implementation ClassB. By taking the interface class into separate package stability is achieved. Here the interface class is more stable than the implementation class.

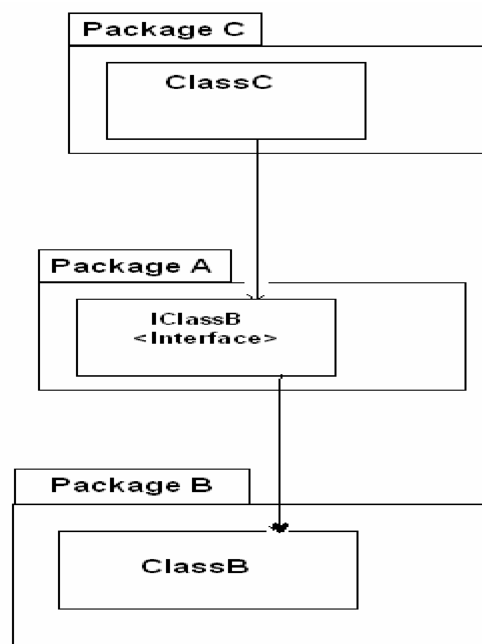

Figure 2.6 Sample application of SAP in UML Notation

The above object-oriented design principles aims to achieve a modular and reusable design. By applying these principles one can achieve easily manageable system software considering the functional behavior.

Object-oriented design principles are good at modularization of the system functional behavior. Whereas it is not the case when we consider the non-functional system requirements that crosscut the software modules.

Such concerns that are scattered along the functional blocks of the system are called “crosscutting concerns”. By using object-oriented programming it is not possible to have a well modularized design for those non-functional concerns. In the next section the not, on of “crosscutting concerns” is described.

2.4 Crosscutting Concerns

Software development is becoming a more complex issue, as the requirements get more complex. The software should handle a large number of wishes, requirements and needs. Software development, therefore have to deal with a large number of concerns. The term “concern” here is used to illustrate any matter of interest.

Some concerns in the software development process are related with the functional requirements of the product itself. However, there are some concerns, which are mostly related to the development process itself. These concerns are usually non-functional concerns. The term “concern” here is used to illustrate any matter of interest.

Separation of concerns is a basic principle of software engineering. The separation of concerns principle comes out from the fact that, dealing with complex problems is only possible by dividing them into simpler sub-problems.

The most well known result of separation of concerns principle is modularization. Each module in the software is designed to deal with only one specific concern. Modular units in system software are easily

manageable and reusable building blocks. Separation of concerns can be said to improve manageability and reusability of the system

By using the predominant object-oriented design techniques, it is possible to separate the functional concerns of the software. An expert software designer can build a modular program handling the functional behavior of the system by using object-oriented design methodology. Whereas, regardless of the programmer's experience, there are some non-functional concerns which using the traditional object-oriented design techniques cannot separate. These concerns crosscut the behavior and implementation of several or sometimes many functional modules of the software. So they are called as "crosscutting concerns". Figure 2.7 shows a typical placement of the software modules in a sample software project.

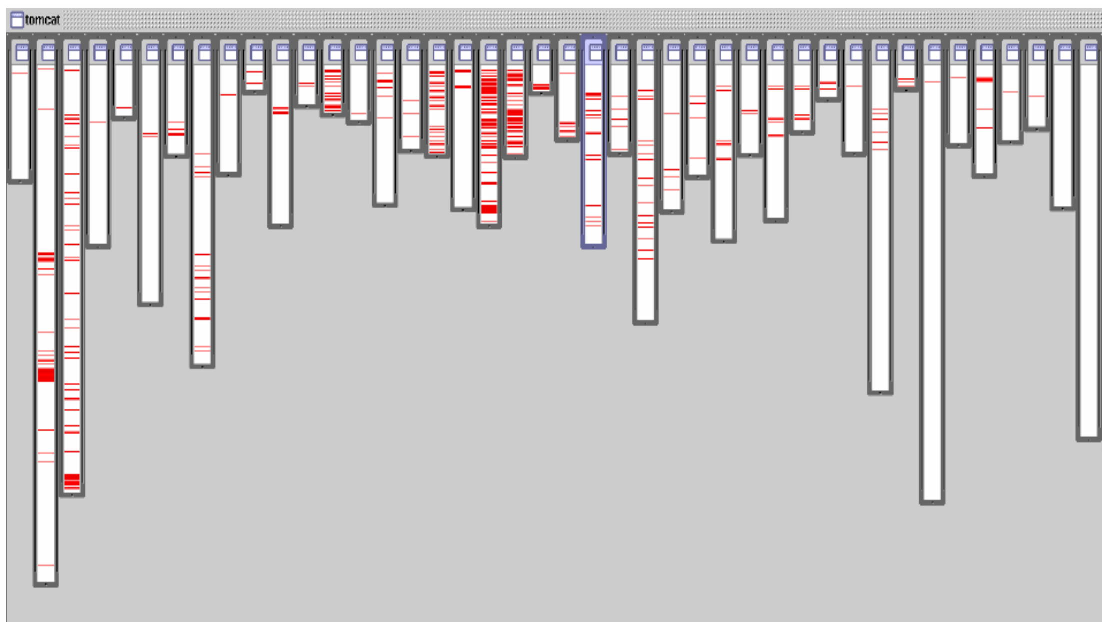

Figure 2.7 Sample Orientation of Modules in a Software Project [11]

The white blocks show the functional building blocks of the project. As seen from the figure the functional blocks are well modularized. Whereas the horizontal lines scattered to the functional blocks. Horizontal lines, showing the logging concern in this example, are scattered along several modules. So the red-colored concern is said to be a crosscutting concern.

Typical examples of crosscutting concerns in the software development process are logging, error handling, memory management and synchronization. For real-time systems, because of the nature of being real-time, timing becomes a crosscutting concern though.

Crosscutting concerns causes two main problems in the software. The first problem is the scattering problem. The design of crosscutting concerns are scattered in several blocks as seen in Figure 2.7 for the logging concern. The second problem is the design of one block becomes dependent to the design of other blocks, which is called “tangling”.

Scattering and tangling makes the system software harder to modularize. The un-modularized concerns preclude the reuse of the functional blocks of the design.

As stated above, traditional object-oriented design techniques are not able to modularize those crosscutting concerns. So it can be said that there is a gap in the object-oriented design process in the field of separation of concerns [12].

The Separation of Concerns problem in the object-oriented design process can be solved by using Aspect Oriented Programming techniques as stated by Rashid and Blair in [13].

2.5 Aspect Oriented Programming

Object Oriented Programming has been introduced as a fundamental technology that aid software engineering since; real domain problems can easily be mapped to an object-oriented domain. In other words it is easy to solve the real-life problems when thinking in an object-oriented manner.

The object-oriented design paradigm is a good design methodology, guiding software towards the solution of real life problems. However, as discussed in the previous section, object-oriented programming is not good at solving issues related to non-functional concerns.

The problems that object-oriented programming is not good at solving are called as crosscutting concerns. Crosscutting concerns, which are introduced in the previous section, are the concerns that crosscut the system functionality.

As stated in [14], Aspect Oriented Programming (AOP) is a programming paradigm that supports the modular implementation of the crosscutting concerns. The software units that are used for modularizing those crosscutting concerns of the system in AOP are called as *"aspects".* Aspects are described in [15] as a piece of code that describes a recurring property of a program.

Aspects provide crosscutting modularity to system software. In other words, programmers can use these units as modular units for crosscutting functionality of the system software.

In the object-oriented design methodology, since the crosscutting concerns of the system are scattered over the functional modular units, they are hard to control. Using aspects in implementing this crosscutting functionality, programmers are able to control the crosscutting behavior of their code more easily. Gregor Kickzales, an aspect pioneer, said that "Programmers could thus think of write, view, edit and otherwise address these issues as a unit, implementing changes or upgrades across all applicable code sections, rather than by having to modify each applicable piece of code."

Aspects thus make the programmers' life easier. By the help of aspects programmers get the power to control the crosscutting functionality in their code. Figure 2.8 shows the use of aspects and their benefits to system modularity from the point of programmer's view.

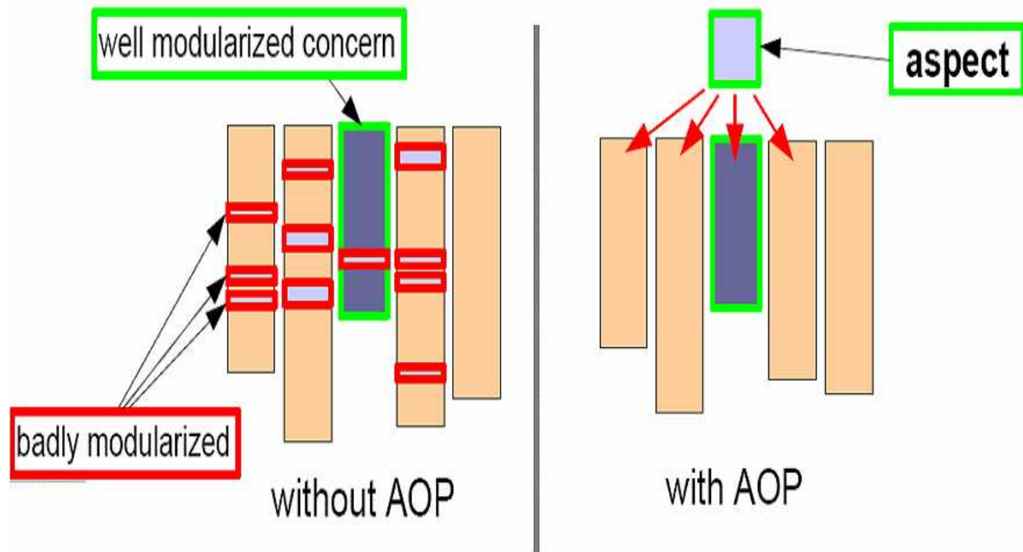

Figure 2.8 Use of Aspect Oriented Programming [24]

The left half of Figure 2.8 shows sample system software designed by traditional object-oriented methodology. The blocks show different functional modules of the system. The highlighted boxes in the blocks show the crosscutting non-functional system code, scattered among the functional modules. The functional blocks that contain many crosscutting code inside are called as badly modularized. This is because, those modules are hard to manage and reuse. They are dealing with more than one system property.

The right half of Figure 2.8 shows the AOP implementation of the same code. In that half the crosscutting functionality of the system is well modularized within aspect code. So, the entire functional modules and the crosscutting functionality of the system are made to be well modularized by the use of aspect-oriented programming.

2.5.1 How Aspect Oriented Programming Works

Aspect-Oriented Programming was first introduced in [16] by Gregor Kickzales. Aspect-Oriented Programming is introduced as an additional patch to the Object Oriented Software Design to solve the modularity problem of crosscutting concerns in system software.

AOP aims to reach, modify and extend the component code of system software without changing any building blocks in the system structure. Aspect Oriented Programming introduces two new elements as tools for software development: *aspect language* and *aspect weaver*.

Aspect language is used to program the aspects and differs from the component language of the system. Aspect language uses some special symbols and wildcards to reach the component code of the system.

Aspect weaver, on the other hand, is used to weave the aspect code into the component code. The aspect code, hung to special locations in the component code, is weaved into the component code by passing them through the aspect weaver. The output of the weaver is a combination of the component code and aspect code. The woven code produces the executable after passing it through a standard compiler.

In fact, crosscutting concerns still exists in the resultant code produced by the aspect weaver. However, from the programmer's point of view we deal with this crosscutting functionality in a more modular way.

The role of aspect weaver in Aspect Oriented Programming is illustrated in Figure 2.9 below.

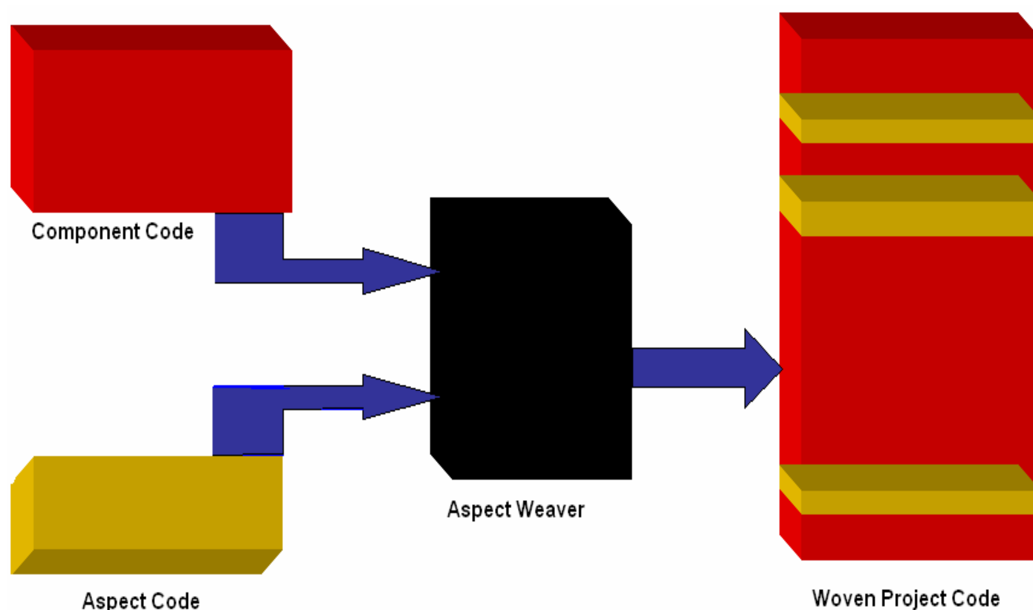

Figure 2.9 Aspect Weaver

In the above figure aspect code is the code segment implementing the crosscutting functionality of the system. The component code on the other hand deals only with the functional behavior of the system.

The operation procedure of AOP is shown with an illustrative example below. The example is a typical implementation of the logging concern taken from the Audio Switch project. Figure 2.10 and Figure 2.11 shows the object-oriented implementation of this concern and figures Figure 2.12 and Figure 2.13 show the aspect-oriented counterpart of the same concern.

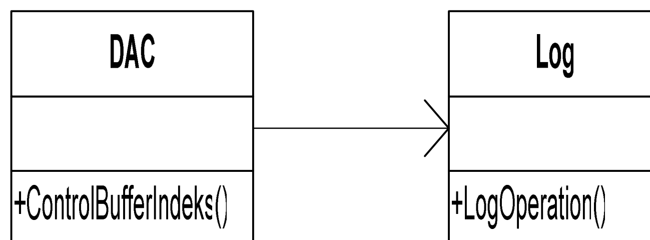

Figure 2.10 Class Diagram of Object-Oriented Implementation of Logging Concern in Audio Switch Project

```
OMBoolean DAC: ControlBufferIndeks() {
   itsLog->LogOperation("DAC: ControlBufferIndeks")
   int iIndeks;
   for(iIndeks = 0; iIndeks < 16; iIndeks++)
     {
       if(itsDACBuffer[iIndeks]->iIndeks != iIndeks)
         {
           return (false);
         }
     }
   return (true);
}
```

Component Code

```
void Log: LogOperation(char* cpOperationName) {
     printf("%s operation is called \n", cpOperationName);
}
```

Component Code

Figure 2.11 Object-Oriented Implementation of Logging Concern in Audio Switch Project

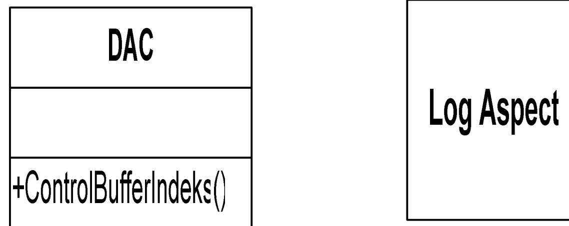

Figure 2.12 Logical Settlement of Aspect-Oriented Implementation of Logging Concern in Audio Switch Project

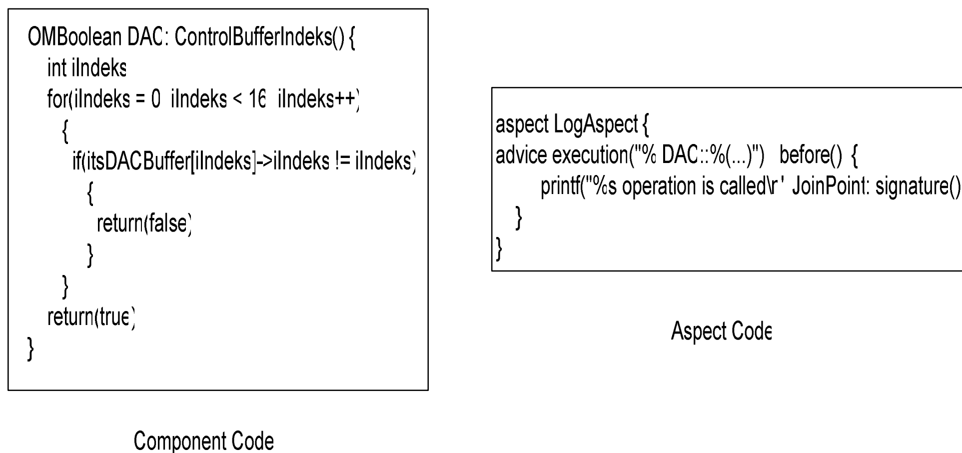

Figure 2.13 Aspect-Oriented Implementation of Logging Concern in Audio Switch Project

For the object-oriented implementation of the logging concern in the Audio Switch project, a “Log” class with a “LogOperation” method is implemented. The “DAC” class uses the logging facility of the “Log” class via message passing. The implementation of the “LogOperation” method of Log class and “ControlBufferIndeks” method of the DAC class are shown in Figure 2.11 above. As seen in Figure 2.11 above the logging concern crosscuts the implementation of DAC class in the object-oriented implementation.

On the other hand the aspect-oriented implementation of the same logging concern is more modular. As seen in Figure 2.12 the Log class in the object-oriented implementation is replaced with a “Log” aspect implementation. The

message passing and the crosscutting behavior of the logging concern are eliminated with the use of advice code given in Figure 2.13. The link between the functional DAC class and the non-functional “LogAspect” aspect is handled by the aspect weaver as given in Figure 2.9.

From the programmers point of view the crosscutting logging functionality becomes more modular and controllable. In the resultant woven project code there is still crosscutting concerns. However, the implementation of the crosscutting functionality differs from the object-oriented implementation in the woven code there is no message passing between software objects for the implementation of crosscutting concerns. A slice of the woven code for the logging concern described in the above figures is given in Figure 2.14 below.

```
OMBoolean DAC: ControlBufferIndeks() {
  AC::ResultBuffer< int > result
  TJP__ZN3DAC19ControlBufferIndeksEv tjp__ZN3DAC19ControlBufferIndeksEv = {
&(TJP__ZN3DAC19ControlBufferIndeksEv::Result&)result }

  AC::invoke_LogAspect_LogAspect_a5_before<TJP__ZN3DAC19ControlBufferIndeksEv> ()
  AC::invoke_DACTimeMeasure_DACTimeMeasure_a0_before ()
  ::new (&result) int (this->__exec_old_ControlBufferIndeks())
  AC::invoke_ErrorHandling_ErrorHandling_a0_after<TJP__ZN3DAC19ControlBufferIndeksEv>
(&tjp__ZN3DAC19ControlBufferIndeksEv)
  return (int &)result

}
```

Figure 2.14 Aspect-Oriented Implementation of Logging Concern

As stated previously, aspect language is a programming language extension that is used to program the aspects separately. For the adoption of Aspect Oriented Programming to software, tool and language support is a prerequisite. There are several aspect-oriented language extensions. The most popular two language extensions of aspect-oriented programming are AspectJ (aspect-oriented extension of Java) and AspectC++ (aspect-oriented

extension of C++). In this thesis work AspectC++ is used in the development phase.

Aspect Oriented programming was predominantly applied in Java. So AspectJ was the predominant aspect-oriented language extension. Since Java language does not respond the requirements of real-time and embedded systems design, the need for an aspect weaver and aspect language extension for C++ appeared.

Adoption of AOP to C++ was late when compared to Java. Since developing a weaver in C++ is a tedious task, the production of the weaver and AspectC++ language was a bit late when compared with AspectJ. But by the studies of aspectC++ research group the fully-fledged AOP support is brought into the C++ domain.

2.5.2 An Aspect Language: AspectC++

Aspect Oriented Programming (AOP) is a programming paradigm that is used to implement the crosscutting concerns in the object-oriented domain. AOP is a popular programming paradigm used in Java language.

Adoption of AOP to C++ was late when compared to Java. Spinczyk et. al. explains the cause of this delay in [14] as the complexity of C++ language. Considering the domain requirements of embedded and real-time systems C++ is more powerful than Java. For systems, for which the run-time and memory efficiency are crucial factors, C++ has nearly no alternatives. The need to a C++ language extension of Aspect Oriented Programming is emerged from this fact.

The AspectC++ research group has designed an aspect weaver for the C++ language and has produced a language extension of AOP for C++ language, namely AspectC++ language.

AspectC++ is an aspect-oriented extension of C++ language; therefore every valid C++ code is also a valid AspectC++ code. AspectC++ brings two new

language elements to pure C++. These new language elements are the “join points” and “advice”. AspectC++ uses “match expressions” and “pointcuts” to form the aspects. These new programming structures are defined in the following paragraphs, but its is worth to note that these structures in the AspectC++ language are very similar to their corresponding structures in the AspectJ language.

A “join point” in AspectC++ is referred to a static location in the program structure. “Advice” is the code segment that affects the static program structure at the join point locations. AspectC++ gives the programmers the ability of hanging advice code to the static program structure at the joinpoints.

AspectC++ defines three types of advice definitions: “code advice”, “introductions” and “aspect order definitions”. The code advice defines the execution time of the advice code by using “before”, “after” and “around” keywords. Programmers can control their aspect code to run before, after or around a specific function in the code structure by using these keywords. These keywords and code advice itself are meaningful only within aspects.

If there are more than one code advice, defined in different aspects, hung at the same join point in the program structure, the programmer can arrange the operation order of those code advice by using the aspect order definitions.

A set of join points defined in AspectC++ is called a “pointcut”. Pointcut expressions are defined by “match expressions”. Match expressions are used to identify the exact place where the pointcut refers. “Named pointcuts” can be defined anywhere in the program, whereas advice can only be defined within aspects. The following example illustrates all these concepts clearly.

```
aspect LogService {
pointcut AllOperations() = % Processor::%(int);
advice execution(AllOperations()) : after() {
cout<< “Operation From Processor Class is invoked” << endl;
}};
```

In the above example, the "*AllOperations()"* pointcut is defined by using match expressions and it refers to all operations in *Processor* class expecting an integer variable and returning any type. The special symbol "%" is a wildcard used in match expressions.

Code advice, that is hung to the join points by using the "*execution*" and "*after*" keywords, is defined to operate after the execution of the operations defined in the pointcut *AllOperations().*

Both the code advice and join point definition encapsulated within a named pointcut is defined in the "LogService" aspect. As stated previously, the programmer has the chance of defining the named pointcut outside the aspect, whereas the "code advice" should be defined within the aspect.

In the following example there is a second aspect defined that has different code advices hung to exactly the same join points in the code structure.

```
aspect TraceService {
advice execution("% Processor::%(int)") : after()
{
   cout<< "Operation From Processor Class is  passed" << endl;
   } };
```

As in the above example match expressions can be used directly in the advice definition. Here the problem is: both the LogService and TraceService aspects contain advices referring to same locations in the code structure. The order of the execution of these two advices can be arranged by using the aspect order definitions. Use of the following order definition advice defined in the *LogService* aspect will operate before the advice defined in the *TraceService* aspect.

```
advice "Processor" : order ("LogService", "TraceService")
```

The above order definition implies that within all of the advice definitions defined in the namespace of Processor Class, the advice defined in LogService aspect will be operated first and the advice code defined in the TraceService aspect will be operated afterwards.

AspectC++ has special keywords and wildcards, used to define match expressions. Moreover programmers have the ability to reach the context information of the structured code within aspects. The keywords and wildcards used to reach the context information and control flow of the code are listed in the AspectC++ Language Quick Reference (see Appendix A).

AOP usage in the implementation of non-functional crosscutting concerns provide several benefits as mentioned in the previous parts. However, since there is not any standardized weaver the, the resultant code produced by the aspect weaver has the potential to produced unexpected errors. Looking at the literature, although some papers point out this possibility, there is not any example of such errors caused of the aspect weaver.

CHAPTER III

IMPLEMENTATION

This chapter describes the Audio Switch project that is implemented to observe the advantages and disadvantages of Aspect Oriented Programming in embedded real-time systems.

Implementation details of the project and the operating environment of the running code are given in this chapter. In the first section, brief information about the run-time environment is given. Implementation details of the project are explored in the following sections.

3.1 Case Study

Audio Switch project is a software implementation of an audio matrix realized in a professional environment. There are forty input channels, each of which can be switched to sixteen different audio outputs separately. The switch can be controlled via a graphical user interface. The user can increase or decrease the signal levels of each input channel. Moreover the user can add a volume offset or completely mute any input channels. Each input channel can be switched to one or more output channels. Besides these, the user has the ability to multiplex several input channels to one or more output channels.

Implemented audio switch is designed as an embedded system. There is a Motorola MVME 5100 main board with a PowerPC 7410 central processing unit. The main board has an internal clock frequency of 400MHz. The project software is running on the real-time embedded operating system VxWorks.

The system's main functionality can briefly be described in three steps. First step is collecting the sampled input data from the input channels of the A/D converter, which are mapped to specific memory locations in the main board's memory. Then the sampled data is processed according to the requirements of the user. The input data is processed in this step and the switching paths are formed. Finally the output signal samples are pushed to the output channels of the D/A converter, which are mapped to specific memory locations in the main board's memory.

Sampling the analog input signal and generating the analog outputs from the sampled data is done by the A/D and D/A converter hardware. The role of the software on this procedure is only in the control level. The sampling rate, discrete or continuous sampling types are arranged by the software.

In the next sections, detailed information about the main board and the operating system VxWorks that the system software is running on is given.

3.1.1 Motorola MVME 5100 Board

As stated in [17] "The MVME5100 Series is the flagship of the Motorola PowerPlus II VME Architecture line, enabling supercomputing levels of performance in a single VME bus slot." Board contains a MPC7410 microprocessor unit with 32 megabyte of cache. Four peripheral mezzanine cards can be connected through 64-bit mezzanine connector.

There is an up to 512-megabyte onboard memory, which can be expanded to 1 gigabyte via memory mezzanines. The internal clock frequency of the board is 400 megahertz.

There are four programmable, 32 bit real-time clocks. There is an Ethernet interface with ten to a hundred megabits per second transfer capacity. The main board can operate with +-5 or +-12 volts voltage ranges. It can operate within 0 to 55 °C temperature ranges.

Motorola MVME 5100 provides booting a variety of operating systems. These operating systems and their producers are listed below.

- VxWorks (Wind River Systems, Inc)
- Integrity (Green Hills)
- Linux (various partners)

Figure 3.1 shows a photo of the card. In this figure, main parts of the board such as VME bus slots, CPU and the PCI expansion slots can be seen clearly.

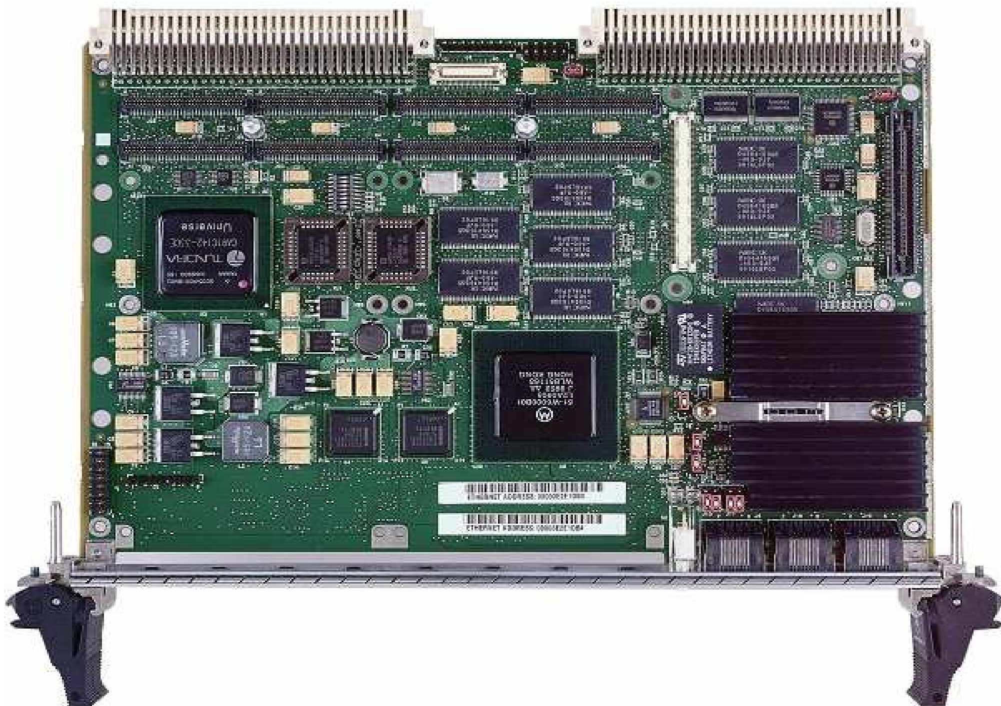

Figure 3.1 Thumbnail of MVME 5100 [17]

A detailed description of the board specifications can be found in Appendix B.

Based on the specifications of the board, it can be said that Motorola MVME 5100 board is suitable for embedded real-time applications.

3.1.2 Real-Time Operating System VxWorks

A real-time operating system (RTOS) schedules multiple tasks according to some initialized priority levels. The tasks follow a predictable operation order. RTOS responds to generated events, almost instantaneously. This property makes an RTOS the ideal control system for mission and time critic applications.

As discussed in the background chapter, real-time systems have some time lines. It is also the case in the Audio Switch project. The A/D converters sample the data with a sampling frequency of 8000 samples per second. That means, in each 125us period of time, new samples are generated and written over to the previous samples. Hence, the system software should be able to collect the samples from all input channels within 125 us. If this time line is missed, several samples are overwritten which causes loss of data. Besides this hard real-time property, the system software should also be able to process all the collected samples within 100 ms. This time limit is not a hard real-time property for the system. Regarding to the human ear's sensitivity, some delay in this processing issue is acceptable.

Since the implemented Audio Switch project has real-time needs a real-time operating system is required and VxWorks is chosen as the operating system of the project.

"VxWorks was created in the early 1980s, when Wind River's founders set out to scale the expertise they'd gathered in the Real-Time Systems Group at the Lawrence Berkeley Laboratory from large physics experiments to device control systems." [18]

The main reasons of choosing VxWorks as the RTOS of the Audio Switch project can be listed as follows:

- There are powerful development tools that make VxWorks easy to configure and use.
- It follows the advances of the hardware evolution with its new releases.

- “It is the most widely used and tested commercial embedded operating system.” [18]

In fact due to the above reasons VxWorks is the most widely used embedded RTOS.

3.2 Project Description

Audio Switch project is a forty-input sixteen-output audio matrix implementation. Switching paths are formed between the input and output channels. The user controls the switch via a graphical user interface. The user can form the switching paths; modify the signal level of any input channels by using the interface.

The project can be examined in three separate blocks. The first block is the A/D Converter block, which is responsible for collecting the audio samples from the predefined memory locations at the main boards memory. Second block is the Data Processing block, which is responsible for forming the switching paths and processing the incoming audio signal samples. The last block is the D/A Converter block, which is responsible for pushing the processed samples to the predefined memory locations in the main boards memory, so that the D/A converter can play the audio correctly. These three blocks and their relations can be seen in Figure 3.2.

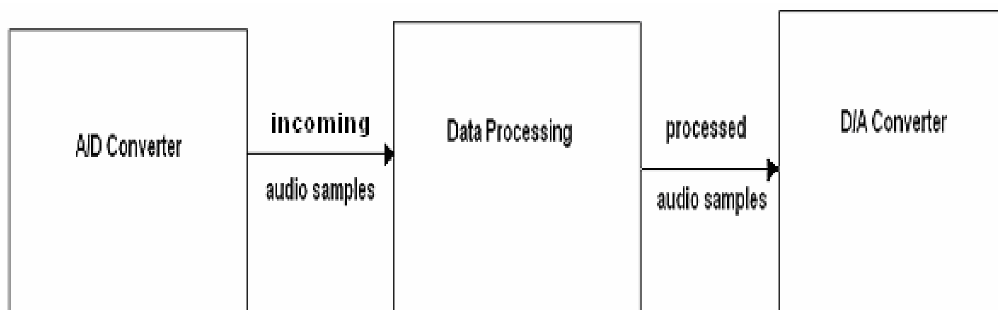

Figure 3.2 Software modules of Audio Switch Project

It is here worth to note that the above blocks are just conceptual blocks; they are not the software packages in the project implementation. In the following subsections these three blocks are described in detail.

3.2.1 A/D Converter Block

The first responsibility of the audio switch is to collect the sampled data of the analog to digital converter hardware within 125us period. This 125us period is a strict time limit for proper operation, because the hardware samples new data in each 125 us period. If the sampled data are not collected in within this time limit, new data are created and overwritten on the previous uncollected samples. This causes a data loss, which is an undesired event.

The A/D Converter block's responsibility is to collect the sampled data without exceeding the timing deadline. As discussed in the previous sections, VxWorks has four programmable real-time clocks. It can also set up interrupts to these real-time clocks. The clock resolutions can be up to 10000 ticks per second. One of these real-time clocks is used to connect an interrupt service routine. The clock resolution is set to 8000 ticks per second so at each 125 us period an interrupt flag is set and the sampled data of the 40 input channels are collected. Then these sampled data is passed to the Data Processing block to be processed and sent to the D/A Converter block according to the set switching paths.

3.2.2 Data Processing Block

The main responsibility of the implemented audio switch is forming the switching paths and processing the input data. The name processing here is adding or subtracting some volume offset, muting some input channels, or increasing the signal level with a multiplicative factor.

There are 40 input data buffers, each of which are kept for a single input channel in the audio switch project. All of the buffers are circular buffers and their sizes are set to keep 1000 samples at a time. These buffers are filled with the A/D Converter block as described in the previous section. Since at each 125 us period, new samples are pushed to the input buffers, the buffers are filled in 125 ms. So it is needed to collect all the samples such that no data is lost because of the overwriting in the circular input buffers.

In order to avoid buffer overflow at each input channel another timer is set to give a triggering event in 100 ms. 100 ms time period is selected regarding the human ear sensitivity as the human ear cannot recognize this much delay.

Within 100 ms time period the system software should be able to collect all the incoming unprocessed samples from the input buffers, modifying them according to the modification factors set by the user and passing them to the D/A Converter block.

User interaction on the whole system operation is centered at the Processing block. The modification factors and switching path information entered by the user at the graphical user interface are taken into consideration at this block. So it can also be considered as the control block of the system.

3.2.3 D/A Converter Block

The last step in the system operation is to push resultant processed and switched data into the memory locations addressed to the digital to analog converter hardware memory. The D/A converter hardware has the capability of collecting the samples put in its addressed memory location at each 125 us period. Within this period the hardware takes new samples and plays audio using those signals. In order to get the correct audio at the hardware output, new samples should be pushed to its addressed memory locations at each 125 us period. If this timeline is exceeded, the hardware plays the same sample more than once, which causes a distortion in the output signal.

In order to avoid the signal distortion the system software should be capable of pushing new samples at each 125 us period. This work is handled in the D/A Converter Block. There are 16 input buffers, one for each output channel, located at this block. The buffer sizes are set to 1000 because of the same reason as the input buffer sizes.

Since 125 us is a strict time limit, the system should complete this work preempting whatever it does at that time. For this purpose an interrupt-triggered operation is needed. Since there is an interrupt set for the A/D Converter block's operation, it is also used to trigger the D/A Converter block's operation. Using the same interrupt the system is arranged to push the processed samples to the predefined memory locations addressed to the D/A converter hardware memory.

With this last conceptual block the system software satisfies its functional requirements. However, besides these functional requirements, there are also some non-functional requirements needed for the system development and reliability. These non-functional requirements can be listed as logging, error handling, memory management and timing (real-time). As it is described in the background chapter, these requirements are the most common crosscutting concerns in embedded system software development. By using object-oriented design techniques it is not possible to modularize these non-functional requirements.

In this thesis, these non-functional requirements which constitute crosscutting concerns of the Audio Switch project, are implemented both using the object-oriented and aspect-oriented programming techniques. These two implementations are compared with respect to both software quality metrics and embedded real-time performance metrics. The comparison metrics and the results obtained from the analyses are given in the evaluation chapter.

Project implementation and the functionality of the A/D Converter block, Data Processing block and the D/A Converter block are summarized in Figure 3.3.

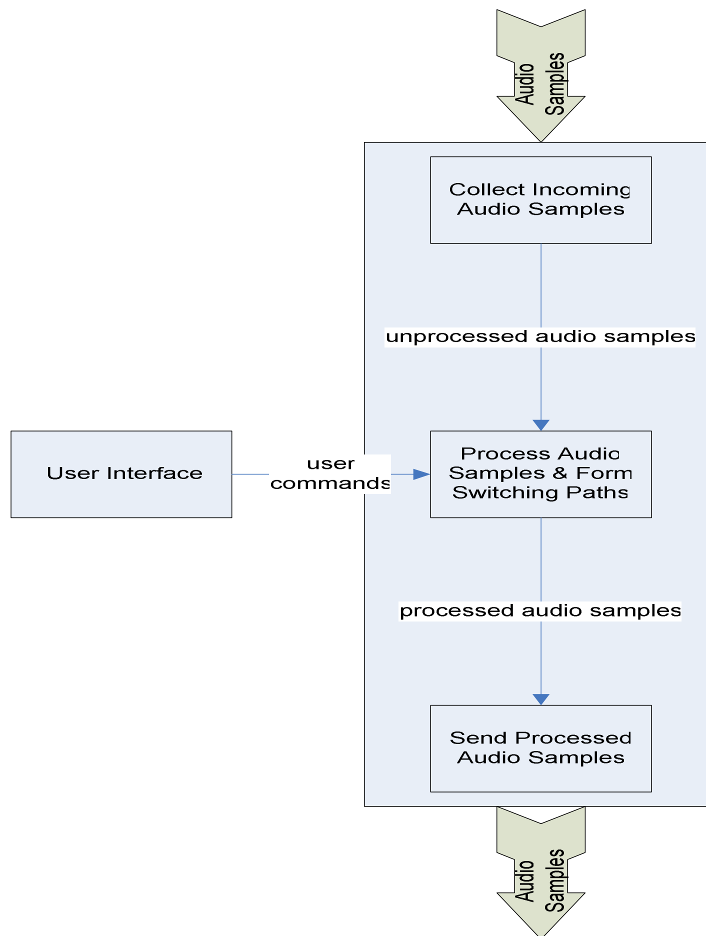

Figure 3.3 Summary of Audio Switch Project Operation

As described in Figure 3.3 audio samples are collected from the input channels of the A/D converter, then these samples are processed according to the user requirements. Then the processed samples are sent to the D/A converter memory to form the audio output. The user interaction is in the data processing block. The switching path formation and the processing issues are done according to the parameters that are set by the user.

3.3 Implemented Non-Functional Concerns

In the Audio Switch project, the most common crosscutting concerns in the software development process are tried to implement. The implemented crosscutting concerns and their descriptions are given in the following sections.

3.3.1 Logging Concern

Regardless of the functional requirements of the implementation, logging is an indispensable need for especially the software development process. It is the mostly known non-functional crosscutting concern in the software development life cycle.

In the object-oriented implementation of this concern in Audio Switch project, this functionality is realized within a class. All the functional classes in the implementation use this functionality, via message calling form the Log class.

The object-oriented implementation of logging concern in the object-oriented domain is shown in Figure 3.4 below.

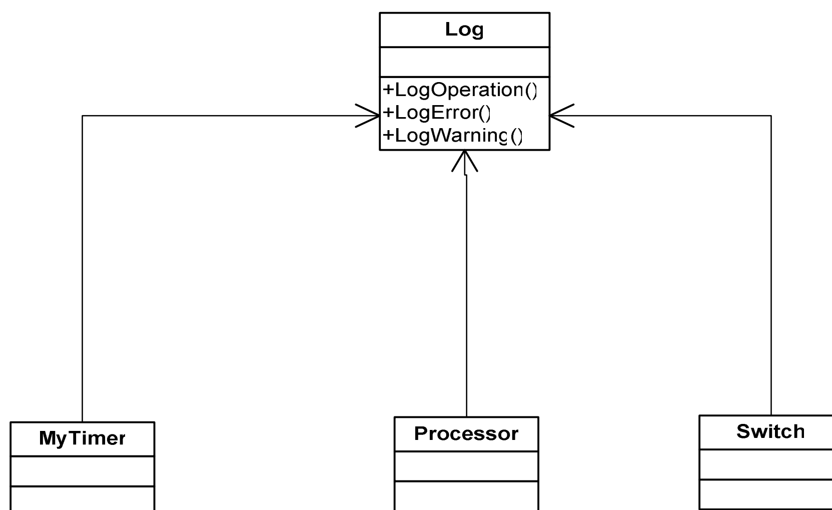

Figure 3.4 Object-Oriented Implementation of Logging Concern

The above figure shows the logging concern in the Audio Switch project for a set of classes in the implementation. In the object-oriented implementation, any change in the Log class has the potential risk of causing changes in the other classes, which have relations with the Log class.

In the aspect-oriented implementation of this concern, the Log class in the above figure is replaced with an aspect. The relations of the other classes to the Log class are broken, and all those relations are handled via joinpoints defined in the Logging aspect. The general implementation of the Logging aspect is given in the following figure for all the classes of the Audio Switch project.

```
aspect LogAspect {

  advice execution("% %::%()") : before(){
     ....;
     ....;
    }

};
```

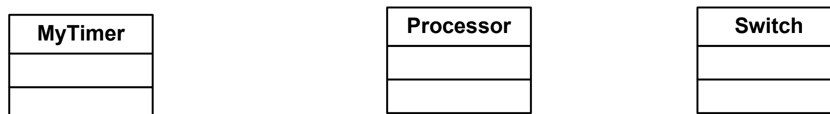

Figure 3.5 Aspect-Oriented Implementation of Logging Concern

As seen from Figure 3.5, the relations of the classes are replaced with the given pointcut expression in the LogAspect. The functionality of Log Class in the object-oriented implementation is handled in the advice code given in the LogAspect aspect.

3.3.2 Error Checking Concern:

Error checking is an important concern, for especially mission critic applications. Error checking in the Audio Switch project handles memory errors and operational errors. This functionality in the object-oriented implementation of the Audio Switch project is implemented in the ErrorChecking class. The classes that do error checking have relations to the ErrorChecking class. In all of the classes where error checking is needed, special error checking statements are inserted into the functional code blocks. The object-oriented implementation is illustrated in Figure 3.6, with a set of classes that need error checking.

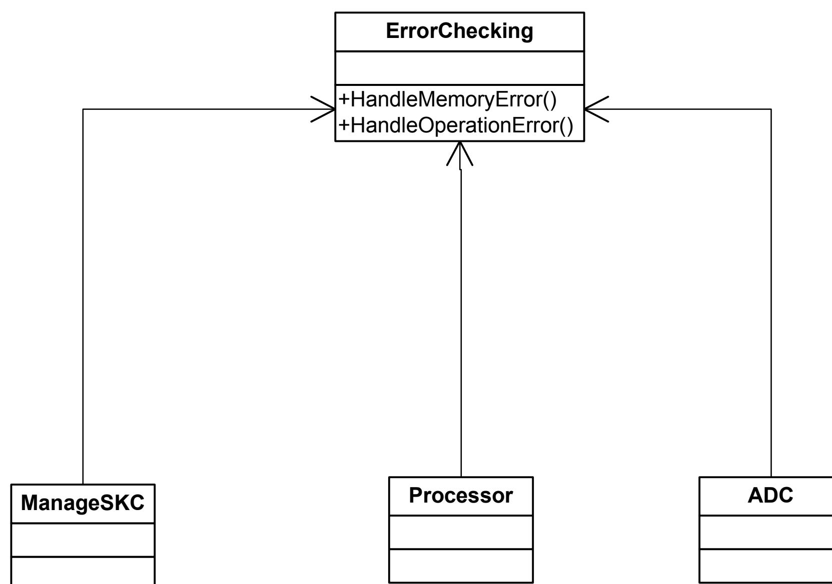

Figure 3.6 Object-Oriented Implementation of Error Checking Concern

In the object oriented implementation, in addition to the inserted error checking code into the functional code blocks, the classes doing error-checking reports the caused errors to the ErrorChecking class via message passing. Hence, any change in this concern causes a series of changes in all of the classes that do error checking.

In the aspect-oriented implementation of error checking, all these issues are handled within a single error checking aspect. The general visualization of the aspect-oriented implementation of error checking concern is give in Figure 3.7 below, with a set of related classes.

```
aspect ErrorHandling{
  advice execution("% %::new(...)") : after(){
    if((*(int*)tjp->result()) == 0)
      ...;
  }
};
```

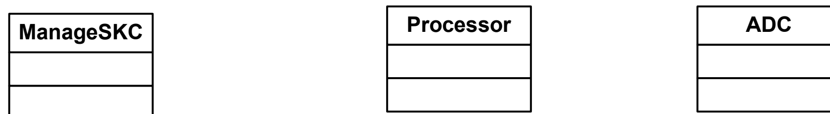

Figure 3.7 Aspect-Oriented Implementation of Error Checking Concern

In the above figure, the aspect-oriented implementation of the error checking concern is shown, just for the memory error checking to provide an opinion about the whole implementation. The relations in the object-oriented implementation are replaced with pointcut definitions in the aspect code. Hence, the error checking concern is made to be more modular.

3.3.3 Range Checking Concern:

Range checking is an application specific concern, which is caused by the needs of the A/D and D/A converter hardware. The converters are capable of processing samples up to 12 bits. The samples, which are grater than this value, cause distortion in the audio data. Range checking is done in order to avoid this distortion. This concern is implemented as crosscutting code

scattered into the functional code blocks in the object-oriented design. An example of the object-oriented implementation is given in Figure 3.8 below.

```
if(itsADCBuffer[iInputIndeks]->LookHOLSample() <= D_MaxSampleVolume)
        ...;
    else
        ...;
```

Figure 3.8 Object-Oriented Implementation of Range Checking Concern

The above figure shows the implementation of range checking concern in DAC class. Any change in the range checking mechanism causes a series of changes in all classes, where range-checking code is scattered. The aspect-oriented implementation of this issue is shown in Figure 3.9 with a small example below.

```
aspect RangeChecking{

 advice (execution("% DACBuffer::PushSample(...)")):before(){
  if(*(int*)tjp->arg(0) > D_MaxSampleVolume)
     ...;
 }
```

Figure 3.9 Aspect-Oriented Implementation of Range Checking Concern

Figure 3.9 shows the aspect-oriented implementation of range checking concern given in Figure 3.8 This implementation makes the range checking concern more modular and easily controllable. Any change in this concern only affects the code in the RangeChecking aspect given in Figure 3.9.

3.3.4 Real-Time Property Concern:

As described in the previous sections, the Audio Switch project has three real-time concerns, two of which are hard real-time. In order to have a control on these real-time properties, run-time of the corresponding operations should be measured. For this purpose, in the object oriented implementation a time measurement class is implemented. The three classes that have real-time specs have relations to this class. The run-time measures and corresponding actions are handled in this class via message passing. The object-oriented implementation of this concern is illustrated in Figure 3.10 below.

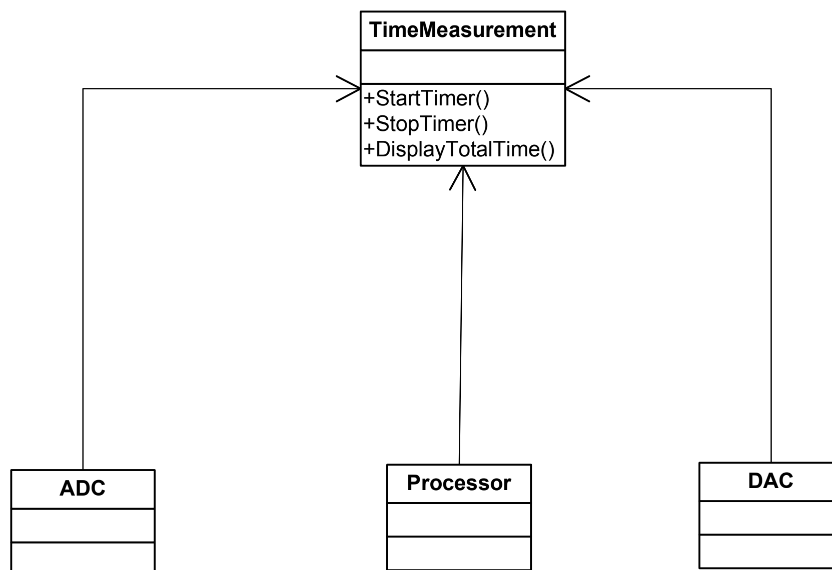

Figure 3.10 Object-Oriented Implementation of Real-Time Property Concern

Any change in the decision procedure in the TimeMeasurement class given in the above figure, will cause a series of changes in the related classes. Hence, object-oriented implementation of this concern is not modular.

The aspect-oriented implementation of the concern is given in Figure 3.10.

```
aspect TimeMeasure {
   advice execution("% Processor::SwitchSample(...)") : around(){
     ...;
     }
               advice execution("% ADC::GetSample(...)") : around(){
     ...;
     }
               advice execution("% DAC::PutSample(...)") : around(){
     ...;
     }
};
```

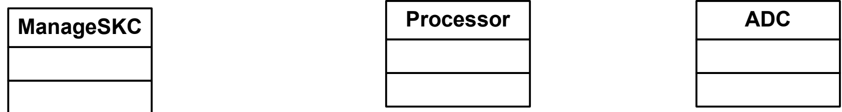

Figure 3.11 Aspect-Oriented Implementation of Real-Time Property Concern

In the aspect-oriented implementation, the responsibility of the TimeMeasurement class is given to the TimeMeasure aspect. The message passing between the functional classes and the TimeMeasurement class in Figure 3.11 is handled by the pointcut definitions in the TimeMeasure aspect above. Hence, real-time property concern becomes more modular by the application of AOP.

CHAPTER IV

EVALUATION

Audio Switch project software is implemented by using the object-oriented design methodology. Unified Modeling Language (UML) is used to model the system software. It is implemented in C++ language, using the Rhapsody design tool. Rhapsody provides support for programming in C, C++ and Ada languages.

As described in the previous chapter, implementation of the functional modules of the system software is done by using object-oriented programming. However, the non-functional crosscutting concerns are implemented both by using the object-oriented and aspect-oriented programming techniques. The two different implementations are compared according to selected software quality and embedded real-time performance metrics.

The crosscutting concerns are implemented and added to the project step by step and at each step resulting implementations are evaluated. This is done to show the impact of AOP by increasing amount of crosscutting concerns in the system software.

Effects of the crosscutting concerns on the metric results do not change with the addition order of the concerns in the project implementation.

In this chapter the evaluation metrics and results of the evaluation process are described. First, the evaluation results from the point of software quality are given, and then the results of embedded real-time performance metrics are explored in the following sections.

4.1 Software Quality

Software quality can be described as the measure of implementation quality of software. Several metrics are proposed to measure the software quality of object-oriented designed system software. These metrics mainly focus on the integration of operation and data to form a system object [19].

Since AOP is based on the existing object-oriented programming concept, software quality metrics are used to evaluate the difference between these two approaches in implementing the crosscutting concerns of the Audio Switch project.

This evaluation process, regarding the software quality, is mainly focused on four system attributes. These system attributes are:

- Reusability,
- Maintainability,
- Understandability,
- Testability.

The above software attributes are in fact, the aims that object-oriented programming desires to achieve.

AOP was previously evaluated in terms of software quality on some desktop computing systems. Most of these studies use the Chidamber and Kemerer (C&K) Metrics Suite to quantify the software quality. As stated in [21, 25] Chidamber and Kemerer (C&K) Metric Suite provides the most comprehensive and best validated set of measures to quantify the software quality.

Because of the above reasons, Chidamber and Kemerer (C&K) Metrics Suite is used. In the following section the C&K metrics and the evaluation results with respect to these metrics are presented.

4.1.1 Chidamber and Kemerer Metrics Suite

C&K metrics suite was generated to fulfill the need for an evaluation metrics suite for Object-Oriented Design methodology. These metrics give numerical results to measure the four software attributes of the system. These metrics are proposed by Shyam R. Chidamber and Chris F. Kemerer in [20] and widely adopted for evaluating the quality of object-oriented system design.

While collecting the metric results aspects are regarded as classes and advices are regarded as the class operations.

The C&K Metric Suite consists of six evaluation metrics. These metrics are:

- Weighted Methods Per Class (WMC)
- Coupling Between Objects (CBO)
- Response For A Class (RFC)
- Lack of Cohesion In Methods (LCOM)
- Depth of Inheritance Tree (DIT)
- Number of Children (NOC)

In the next sections, the metric descriptions and the evaluation results are given in detail. The metric results are taken by using the “Understand for C++” tool produced by the Scientific Toolworks Inc. Understand for C++ is a metric measurement tool for C++ source code. It evaluates the software according to the C&K metrics suite and produces numerical results.

4.1.1.1 Weighted Methods Per Class (WMC)

WMC is the measure of total method complexities of a class. Complexity of a method is calculated with respect to the usage of loops, and conditional statements within that method. Loops and conditional statements increase the method complexity. If a class has n methods with complexities of c1 … cn, then the metric gives a result of “WMC = ∑ci” [20].

WMC is designed to measure the understandability, reusability and maintainability of the software. [20]

First of all the WMC metric results are computed for the object-oriented implementation of the software without any non-functional crosscutting concerns. Then real-time property concern, error and range checking concerns and finally the logging concern are added respectively.

The result of WMC metric is given in Figure 4.1.

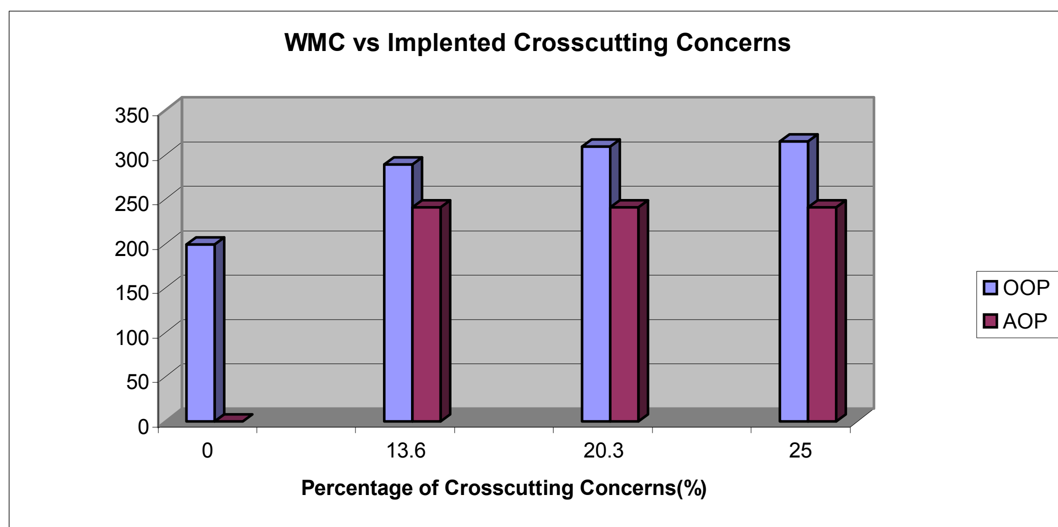

Figure 4.1 WMC Metric Results

The above figure shows the change in WMC metric with the increasing crosscutting concerns lines of code percentage in the project code. The above WMC metric results are the total for all classes present in the system software.

Since AOP defines the crosscutting functionality of the system in aspects it is expected to decrease the total number of classes and operations in the system software. Moreover, since it is possible to give advices to many classes within one aspect, this will therefore decrease the number of tangled methods in a class and decrease the method complexity.

The decrease in the number of methods and method complexity by applying AOP in the implementation of crosscutting concerns, result in an observable decrease in the WMC metrics of the system. Decrease in WMC method will

cause an improvement of system reusability, understandability and maintainability, as stated in. [20]

4.1.1.2 Coupling Between Objects (CBO)

Coupling between two classes is defined as the use of methods of instance variables of a class within another class. [20] If a class is connected to another class with a relation, rather than inheritance, that means these two classes are coupled.

CBO of a class is the number of other classes that are coupled. Coupling between classes hinders modular design and prevents reuse of the system objects. Low coupling implies better design. CBO is a measure of system reusability, understandability, maintainability and testability.

First of all the CBO metric results are computed for the object-oriented implementation of the software without any non-functional crosscutting concerns. Then real-time property concern, error and range checking concerns and finally the logging concern are added respectively.

CBO metric results for the Audio Switch project are taken as the total measure of all classes within the system software. The metric results comparing the aspect oriented and object oriented implementation is given in Figure 4.2.

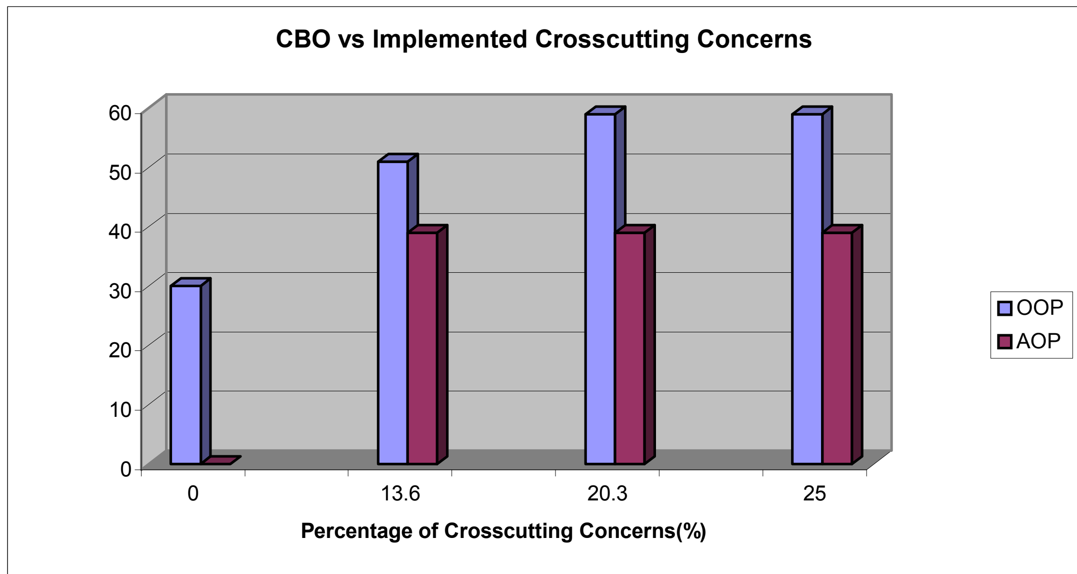

Figure 4.2 CBO Metric Results

Implementing crosscutting concerns with AOP is expected to decrease the coupling between the functional objects. However, another type of coupling between the functional objects and the defined aspects are occurred. This type of coupling does not increase the CBO metric, because no methods or instance variables of the aspects are called within the functional objects of the project. So, application of AOP, in the implementation of crosscutting concerns, is expected to decrease the coupling between system objects.

In Figure 4.2, the decrease in CBO metric of the Audio Switch project, with the application of AOP, is shown. This decrease implies a more modular, reusable and understandable system design. In other words, application of AOP is said to improve the software quality.

4.1.1.3 Response For A Class (RFC)

RFC is defined as the response set of a class, where the response set is the set of methods than can potentially be executed in response to a message received. [20]

RFC includes the messages, outside of the class, that can be invoked by that class. Since, in object-oriented programming, the interaction between the objects is done via message passing, RFC also measures the potential communication between a class and the other classes in the software.

RFC designed to measure the system understandability, maintainability and testability. [20] It is hard to test, understand and reuse a class with a high RFC value.

Increasing RFC implies increasing software complexity, which makes the class harder to test and reuse. AOP is expected to decrease the RFC results of the software by reducing the number of called methods within a class.

First of all the RFC metric results are computed for the object-oriented implementation of the software without any non-functional crosscutting concerns. Then real-time property concern, error and range checking concerns and finally the logging concern are added respectively.

The RFC metric results of the Audio Switch project are given in Figure 4.3

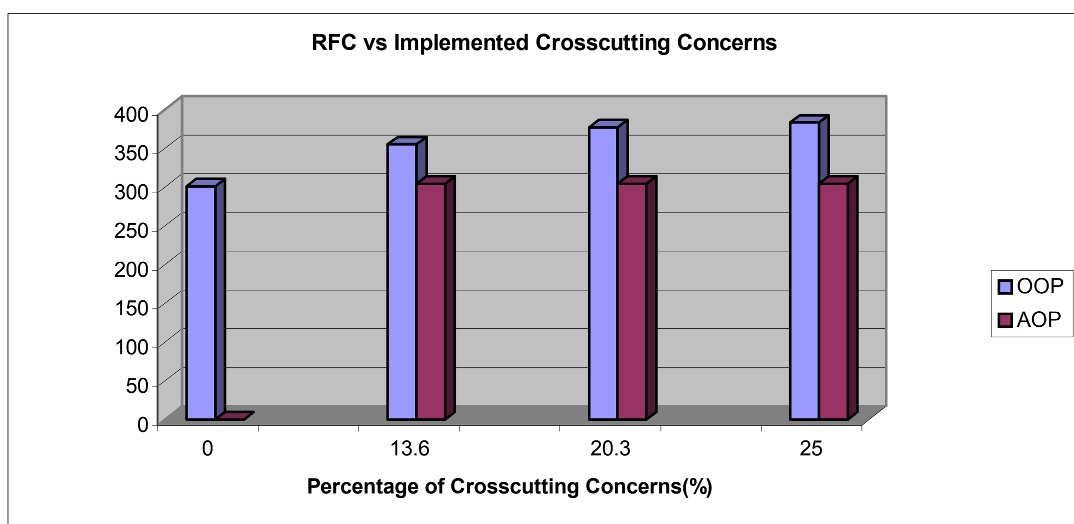

Figure 4.3 RFC Metric Results

Implementing the crosscutting concerns with AOP is expected to result in a decrease in the number of methods that can be invoked in response to a received message. This is simply because the method calls used to

implement the non-functional crosscutting functionality of the system are replaced with less advice code.

Decrease in the RFC metric of system software implies a more reusable and easily understandable implementation. Since, application of AOP in the implementation of crosscutting concerns results in a decrease in the systems total RFC metric, AOP can be said to increase the understandability and testability of the software.

4.1.1.4 Lack Of Cohesion In Methods (LCOM)

Cohesion of an object can be defined as the measure of a class' concentration on doing a job. A class, which is responsible for more than one specific job, is said to be low cohesive. High cohesion implies reusability, maintainability and testability of the system software.

LCOM metric measures the number of methods, within a class, which has no similarity. The similarity between the instance methods of a class is measured by looking at the invoked functions within those methods. [20] If the intersection of the sets of invoked functions of two instance methods is null, then it implies a low cohesive object implementation.

First of all the LCOM metric results are computed for the object-oriented implementation of the software without any non-functional crosscutting concerns. Then real-time property concern, error and range checking concerns and finally the logging concern are added respectively.

The LCOM metric results of the Audio Switch project are given in Figure 4.4.

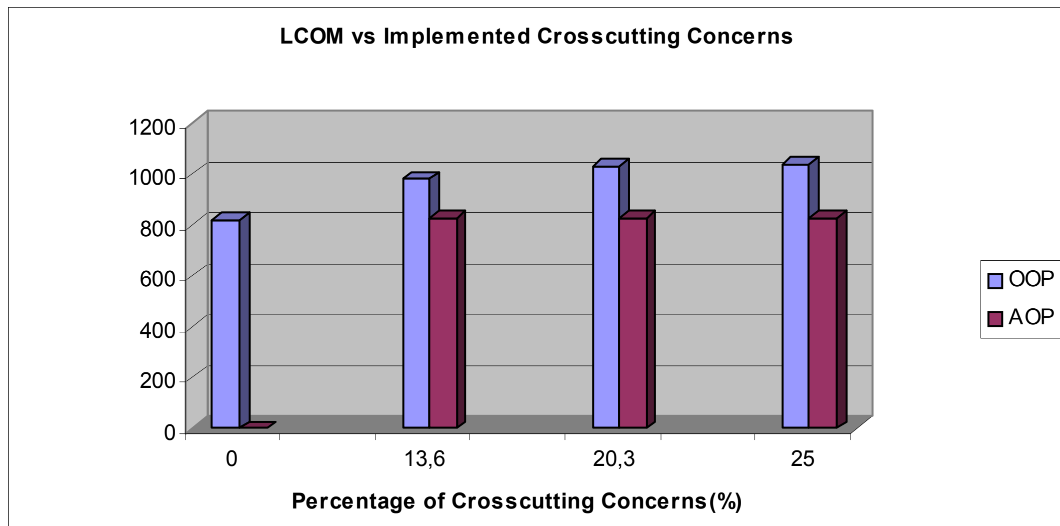

Figure 4.4 LCOM Metric Results

Since AOP modularizes the crosscutting concerns in the system software, the crosscutting responsibilities of the functional objects in the system software are eliminated. This elimination is expected to cause a decrease in the LCOM metric. A decrease in the metric implies more cohesive object implementation in the system software.

As seen in Figure 4.4, the impact of AOP is observed as expected in the Audio Switch project. In other words, AOP improves the reusability, maintainability and testability of the Audio Switch project.

4.1.1.5 Depth of Inheritance Tree (DIT)

DIT is the maximum number of steps from the node to the root of the inheritance tree. [20] A root node has a DIT measure of 0. DIT is a measure of the number of ancestor classes that can affect a class.

DIT is a measure of understandability, reusability and testability. If a class has a high value of DIT measure, it means that class has a large number of inherited methods, which makes the behavior of the method harder to predict. Deeper trees in the software imply design complexity.

Number of inherited methods within a class makes the reuse of the class more difficult. So it is desired to keep a low DIT value for reusable software.

First of all the DIT metric results are computed for the object-oriented implementation of the software without any non-functional crosscutting concerns. Then real-time property concern, error and range checking concerns and finally the logging concern are added respectively.

DIT measurements of both implementations give the same outputs. Since there are no interface classes used in the implementation of the non-functional crosscutting concerns in the Audio Switch project, DIT measures are not applicable for the comparison of the two implementations. However, we can say that, for Audio Switch project, using AOP in the implementation of crosscutting concerns does not increase the DIT measures at all.

4.1.1.6 Number Of Children (NOC)

NOC is defined as the number of immediate subclasses subordinating to a class in the class hierarchy. It is a measure of the number of classes that will inherit the methods of the parent class.

NOC, as the DIT does, measures the understandability, reusability and testability of the software. Greater number of subclasses shows an improper use of inheritance and sub-classing. If a class has a large number of subclasses, that class requires more time for testing. In other words, more subclasses imply the increasing complexity of the parent class, which makes the testability of the class harder.

First of all the NOC metric results are computed for the object-oriented implementation of the software without any non-functional crosscutting concerns. Then real-time property concern, error and range checking concerns and finally the logging concern are added respectively.

NOC measurements of both implementations give the same results. This is because there are no subclass implementations in the implementation of the nun-functional crosscutting concerns of the project software. Hence, NOC is said to be un-applicable for the comparison of the software quality of the

Audio Switch project. However, we can say that AOP can improve the results of the NOC measurements if a subclass in the inheritance tree has a non-functional crosscutting property.

4.1.2 Software Quality Results Summary of Audio Switch Project

Looking from the software quality point of view, the below table, drawn considering the works in [20], shows the mapping of the C&K Metrics on the general software quality attributes and the AOP improvements.

Table 4.1 Mapping of C&K Metrics on Software Quality Attributes

	WMC	CBO	RFC	LCOM	DIT	NOC
Understandability	X	X	X		X	
Reusability	X	X		X	X	X
Maintainability	X	X	X	X		
Testability		X	X		X	X

Table 4.2 Total Effects of AOP on Software Quality Metrics

	WMC	CBO	RFC	LCOM	DIT	NOC
Percent AOP Improvement on the whole project	24%	34%	20%	20%	0%	0%

Table 4.3 Effects of AOP Usage on Software Quality Attributes

	Affect of AOP Usage
Understandability	Improved
Reusability	Improved
Maintainability	Improved
Testability	Improved

Considering Table 4.1, Table 4.2 and Table 4.3 it can be said that; using AOP techniques in implementing the crosscutting functionality of C++ based embedded real-time systems improves the system software quality. Hence AOP usage in embedded real-time systems can be thought as an alternative in the implementation of crosscutting concerns.

4.2 Embedded Real-Time System Performance

The domain of embedded real-time systems is dominated with resource constraints. Especially memory usage and run-time are the main restrictions that shape the embedded real-time software development. To cope with these restrictions, embedded real-time software developers avoid using the structured software design techniques. Hence, most embedded real-time applications are developed in C language.

Object Oriented Programming is still less in demand as some of its concerns like message passing and use of instance variables cause non-negligible performance costs. Software quality metrics such as reusability and understandability and the phenomenon of separation of concerns are considered to be less important [22].

Since AOP reduces some of the overheads of OOP, AOP can provide a performance increase in the field of embedded real-time applications. In this section and the following sections, embedded real-time performance comparison of OOP and AOP in the implementation of non-functional

crosscutting concerns is presented. The performance comparison results of the Audio Switch project are given in the following sections.

The comparison is done on three comparison metrics, which are:

- Memory Usage
- CPU Usage
- Run-Time

In the following sections, detailed description of the metric results, gathered from the Audio Switch project's software are given.

4.2.1 Memory Usage

Memory usage of a program can be viewed as the static and dynamic memory usages. Static memory usages can be easily measured by looking in to the linker map file of the object code. Whereas, dynamic memory usage tests should be performed on the running targets.

Since the dynamic memory usage is the critical issue for embedded systems, in this thesis dynamic memory usage of the Audio Switch project is measured.

Memory usage tests are done on the running target to observe the dynamic memory usage of the two implementations with the increasing amount of crosscutting concerns in the project code. The measurements are done using a special spy agent running on VxWorks.

The memory usage test results are given in Figure 4.5. Memory usage percentages of the two different implementations are plotted with respect to the increasing amount of crosscutting concerns in the project code.

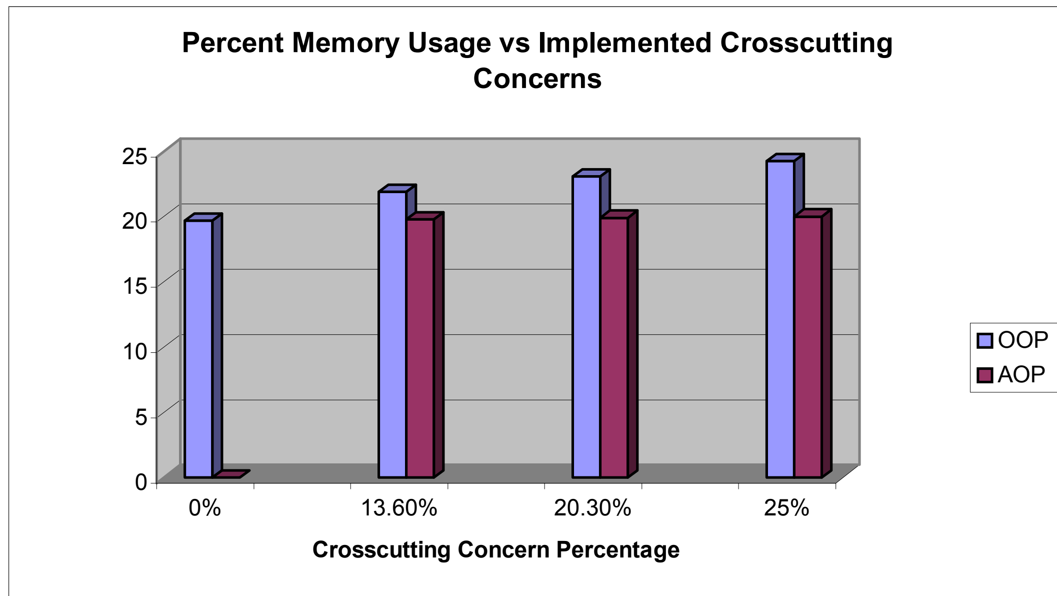

Figure 4.5 Dynamic Memory Usage Results

The main reasons of the memory increase in OOP are the usage of virtual functions as they both increase the size of the caller and the called side; dynamic data structures and global instance construction [22]. Relations are also an important reason for the increase in the memory usage.

Since, implementation of the crosscutting concerns within aspects decreases the use of virtual function declarations and global instance creations, the memory usage of the AOP version of the Audio Switch project is smaller than the OOP version.

Increasing amount of crosscutting code will result in an increase in the memory usage differences between the two implementations. Looking at the results, given in Figure 4.5, it can be said that AOP decreases the dynamic memory usage of the running code, so it seems to be suitable for the embedded applications from the point of memory requirements view.

4.2.2 CPU Usage

CPU usage of a task can be defined as the percentage of the CPU resources that are assigned to the running task to complete its responsibilities. Since

the running system code is composed of parallel running tasks, CPU usage of the total project code can be defined as the cumulative CPU usages of the running tasks.

Regarding the resource constraints of the embedded systems, CPU usage is an important performance criterion. Increasing CPU usage means increasing cost in embedded system design. Implementing a project code, doing the same job, with a lower CPU usage is a need for embedded real-time systems.

Object Oriented Programming, when compared with procedural programming, has some degradations in the CPU usage performance of the system. In other words OOP needs more CPU percentage to satisfy the same requirements than the traditionally programmed counterpart.

The performance degradation of the OOP in terms of CPU usage is mainly caused by the use of virtual function calls and message passing between two parallel running tasks. Message passing between two tasks needs a context switching operation to save the attributes and current state information of the running task, which obviously needs CPU usage.

Implementing crosscutting concerns using AOP is expected to result in an improvement in the CPU Usage performance of the system. Since the AOP reduces virtual function calls and message passing, there is an obvious decrease in the CPU usage percentage of the running code.

The results of CPU usage metric, taken from the Audio Switch project are given in Figure 4.6 below. The results are given with respect to the increasing percentage of the crosscutting code in the running system.

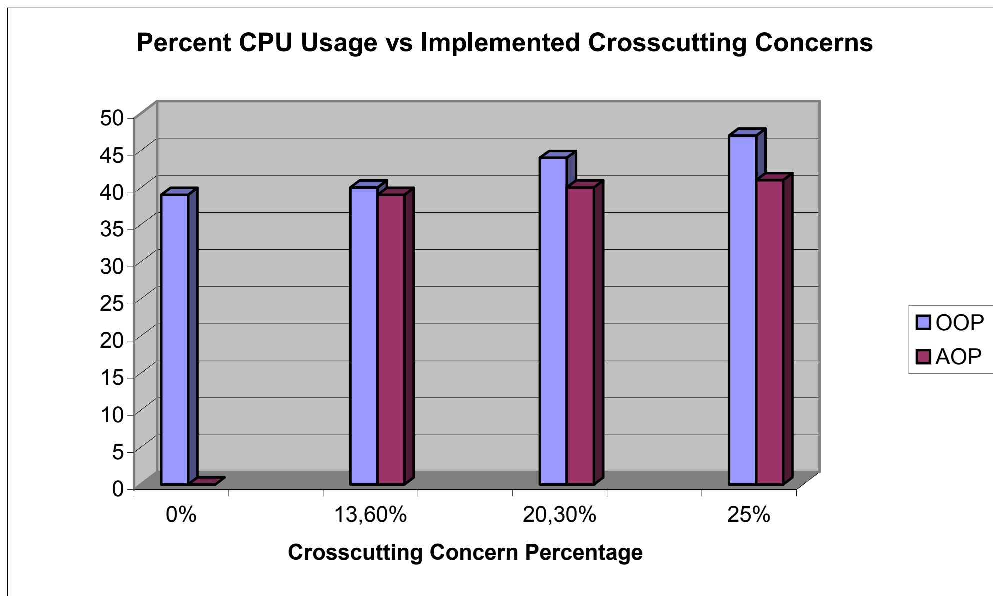

Figure 4.6 CPU Usage Results

As seen in the above graph, AOP decreases the CPU usage of the system code. It is here worth to node that both implementations do exactly the same process and satisfy the requirements of the system.

With the increasing percentage of the AOP implemented code, the effects of AOP become more significant. So we can say that, AOP improves the system's CPU usage performance. Moreover as the AOP implemented code percent increases in the total project code, the CPU usage performance improvements become more observable.

4.2.3 Run-Time

As discussed in the previous chapters, run time has critical importance for real-time systems. Especially for hard real-time systems, the timing deadlines are strict deadlines. Lags in the output production times are not tolerated in real-time systems. Mostly a lagging output means an incorrect output for those systems.

As in all real-time systems, run-time is significantly important for the Audio Switch project. As discussed in the Chapter |||, the system has three real time constraints, two of which are hard real-time constraints.

Consequent runs of the same application can give different results in terms of run time in nearly all of the operating systems. So, for real-time systems, the time measurements are generally taken as the worst case running time of multiple iterations. Because of this reason, the worst-case run-time of the processes of the Audio Switch project is measured. In addition to the worst case measurements the average run time measures are taken in order to give a feeling about the general operation time of the processes.

The measurements are taken by using the “high resolution time stamping” property of the VxWorks operating system. So it could be possible to take measurements in 0,01 us resolution.

Run time measurement results of the Audio Switch project are given in the next sections. First, the two hard real-time jobs run times are given, then the soft real-time job measurements presented in the following section.

The run-time measurements are measured over 100 runs of the project code. So the worst-case run-time measurements give the maximum run time of those 100 runs.

4.2.3.1 A/D Converter Block Run-Time Results

As described in the implementation chapter, A/D Converter Block of the Audio Switch Project has a hard real-time property. The incoming audio samples from the A/D Converter hardware should be collected within 125 us period, in order to prevent overwriting problems and data loss.

The average and worst case run-time results of this hard real-time property are given in Figure 4.7 and Figure 4.8 below.

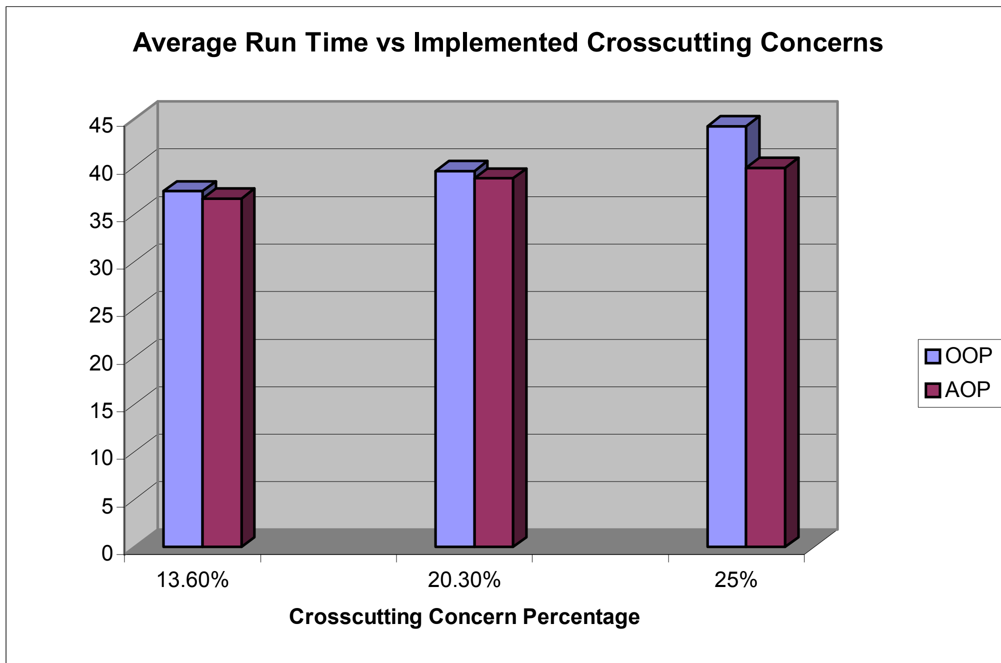

Figure 4.7 Average Run-Time Measurement Results of A/D Converter Block

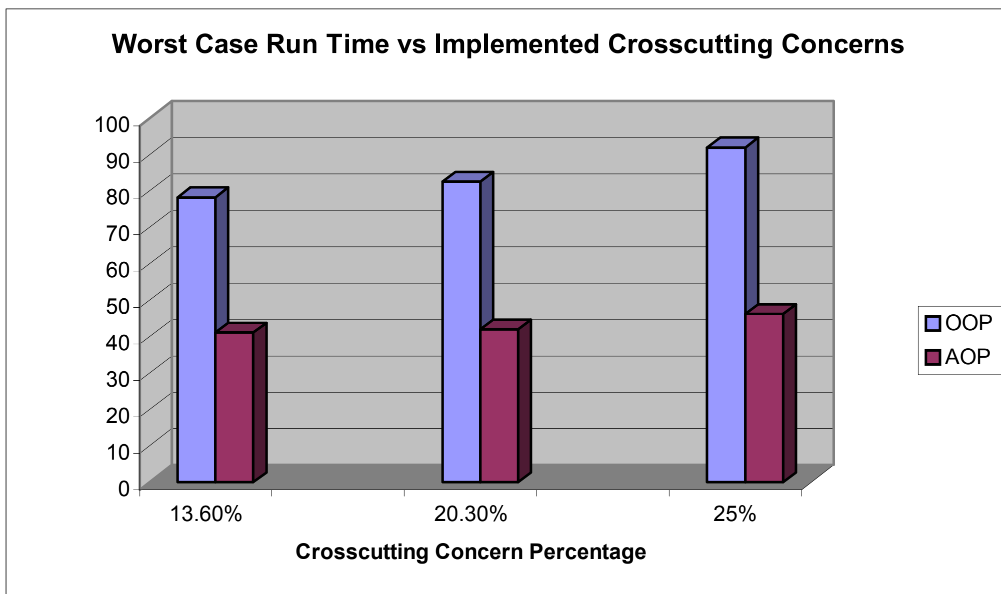

Figure 4.8 Worst Case Run-Time Measurement Results of A/D Converter Block

As seen from the above two graphs AOP usage in the implementation of crosscutting concerns of the Audio Switch Project resulted in a decrease in the run-time. Especially the worst-case run-time decreases significantly. This

decrease in the run-time can is mainly caused by the decrease in the use of virtual functions and message passing between the software modules.

4.2.3.2 D/A Converter Block Run-Time Results

D/A Converter block has also a hard real-time responsibility. This block should be capable of pushing the processed audio samples to the addressed memory locations of the D/A Converter hardware in 125 us period. For a successful operation and prevention of data loss the operation time should not exceed the time line.

The average and worst case run-time results of this hard real-time property are given in Figure 4.9 and Figure 4.10 below.

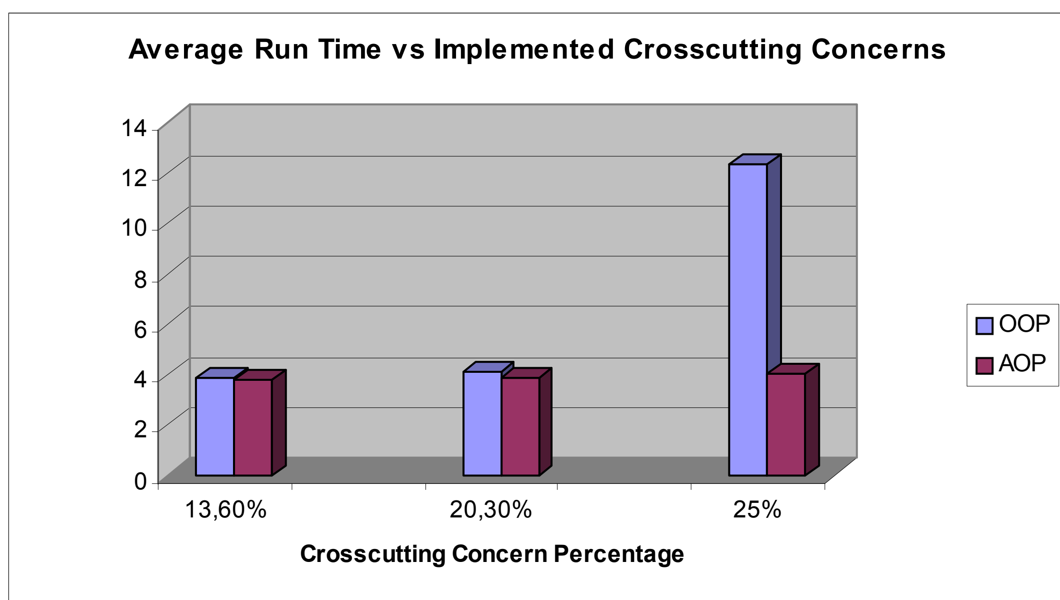

Figure 4.9 Average Run-Time Measurement Results of D/A Converter Block

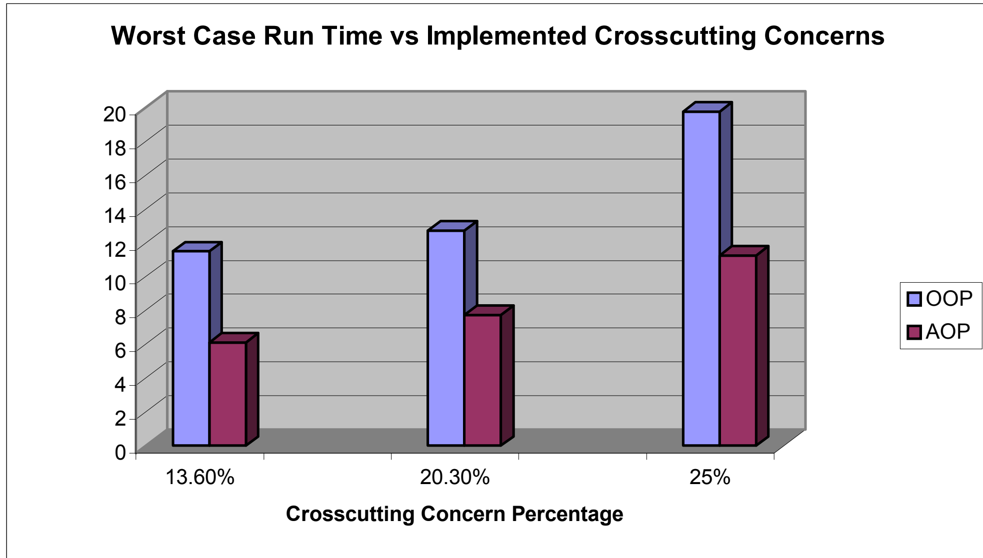

Figure 4.10 Worst Case Run-Time Measurement Results of D/A Converter Block

The code of this block does not have much responsibility, so the percentage of crosscutting code to the total lines of code for this block is nearly a half. So the impact of AOP on real-time systems can be seen clearly in the above graphs. Especially the worst-case run-time differences show the improvements of using AOP significantly. As explained in the previous section the decrease in the run-time is mainly caused of the decrease of virtual function calls and message passing.

4.2.3.3 Data Processing Block Run-Time Results

Data Processing block has a soft real-time responsibility that it should complete within 100 ms period. This block, as described previously, has the responsibility of data modification with respect to the user needs. Moreover this block is also responsible for collecting all the samples buffered in the incoming A/D data buffers and pushing all the processed samples into the outgoing D/A data buffers within this time period.

The average and worst case run-time results of this hard real-time property are given in Figure 4.11 and Figure 4.12 below. Since the operation time is in the order of ms the results are given in ms resolution.

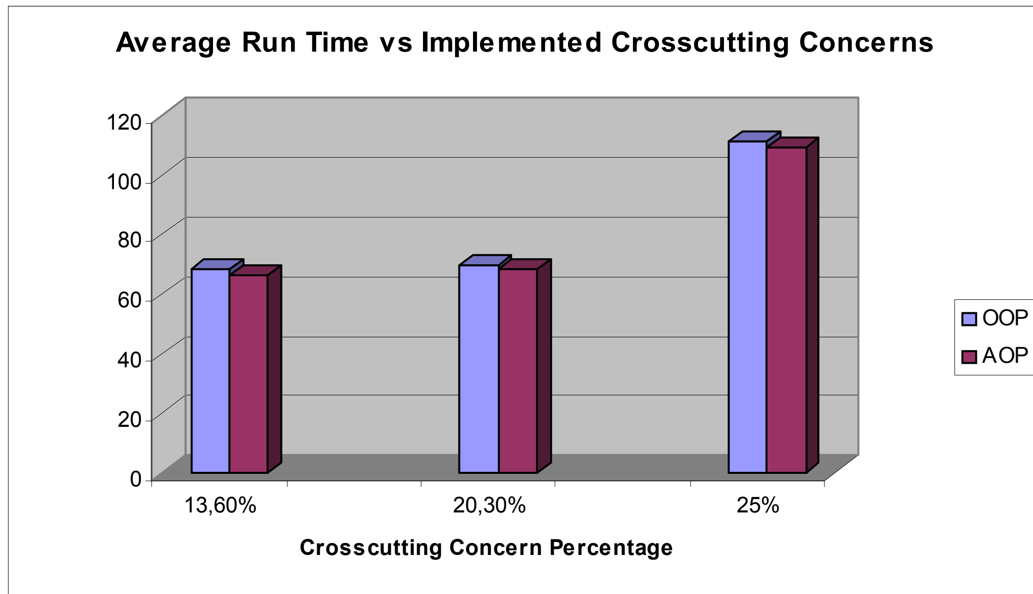

Figure 4.11 Average Run-Time Measurement Results of Data Processing Block

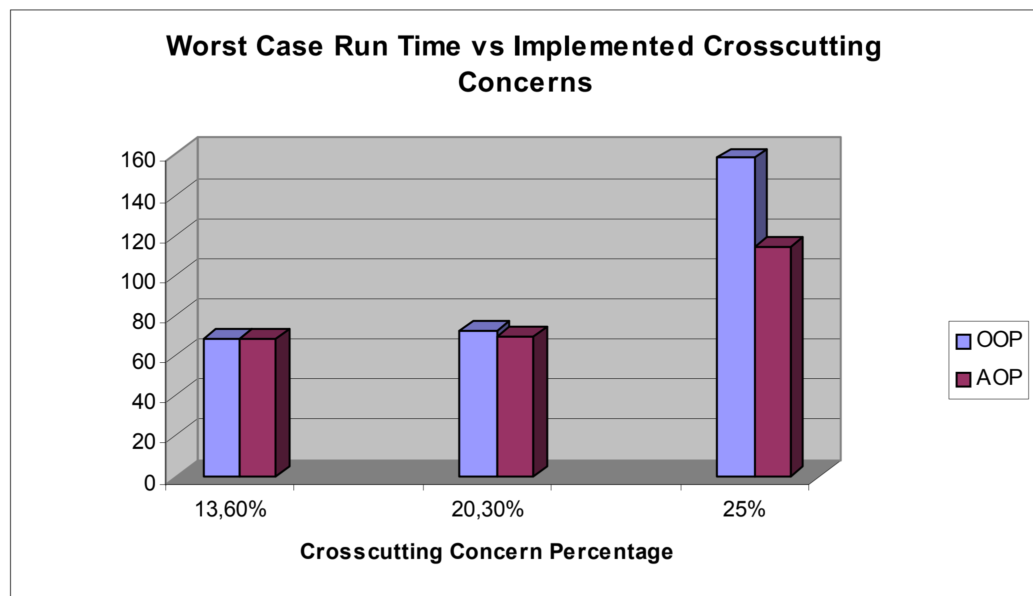

Figure 4.12 Worst Case Run-Time Measurement Results of Data Processing Block

The Data Processing block has the most time consuming responsibility set among the three logical blocks of the Audio Switch project. The percentage of the crosscutting code within this block is less when compared with the other two building blocks. So the average run-time difference between the twp implementations is not as significant as the previous results. However, when we look at the worst-case run-time results, there is a point that is not observed in the previous block measurements. When the crosscutting concern's lines of code percentage in the project code reaches to 25% the worst case run-time of the Data Processing Block exceeds the 100 ms time line, which causes a delay in the transfer of the processed data. Since the incoming and outgoing data buffers are large enough this much lag does not cause a data loss. However the AOP version gives reasonable run-time measurement results lowers the effect of the delay.

Looking at the run-time of the above three blocks, it can be observed that AOP provides an improvement in the run-time metrics of the software. Moreover looking at the differences between the average and the worst-case run-time results, we can say that AOP usage makes the system less variable in terms of run-time. This is because of the variable delays caused in case of message calls to satisfy the crosscutting functionality of the system. Since run-time is a critical issue for real-time systems, application of AOP seems to have beneficial results in embedded real-time systems.

4.3 Summary of Embedded Real-Time System Performance

The embedded real-time performance results of the Audio Switch project are given in the previous sections.

Application of AOP provides a decrease of 4% in the dynamic memory usage of the running application, when the crosscutting code is 25% of the entire system code. From the metric results on dynamic memory usage, it can be concluded that effects of AOP is increasing with the increasing use of aspects in the system code.

The CPU usage results show that, aspect usage in the implementation of crosscutting concerns provides a decrease in the CPU usage of the entire software. In other words, with the application of AOP, the system satisfies its requirements with a lower CPU usage.

AOP also has an improvement in terms of run-time. Especially, effects of AOP can be seen clearly in Figure 4.12, where the AOP implemented code and OOP implemented code percentage are nearly same. So, it can be said that AOP can be used to code a faster algorithm in embedded real-time systems.

The embedded real-time metric results of the Audio Switch Project are given in Table 4.4 and Table 4.5 below. Table 4.4 gives the metrics results for the project, in which all the mentioned crosscutting concerns are implemented.

Table 4.4 Effects of AOP on Embedded Real-Time Metrics

	Memory Usage	CPU Usage	Worst-Case Run-Time A/D Converter Block	Worst-Case Run-Time D/A Converter Block	Worst-Case Run-Time Data Processing Block
Percent AOP Improvement on the whole project	25%	15%	50%	64%	38%

Table 4.5 Embedded Real-Time Metrics Metrics Summary

	Effect of AOP Usage
Memory Usage	Decreased
CPU Usage	Decreased
Run-Time	Decreased

To sum up, it can be said that AOP usage in the implementation of crosscutting concerns in embedded real-time systems provides an improvement in terms of embedded real-time performance metrics.

CHAPTER V

CONCLUSION

Separation of Concerns is a key concept in software development. Object Oriented Programming is good at modularizing the functional behavior of the systems. However it has some problems in crosscutting concern modularization. Aspect Oriented Programming, which is developed over the existing OOP concepts, can be used to circumvent these problems of the OOP. In addition to crosscutting concern modularization, the embedded real-time performance improvements make AOP a solution to solve the performance overhead problem of OOP. Hence AOP can be a more suitable development methodology for embedded real-time systems.

In addition to above contributions, this study shows the power of AspectC++ as an AOP language that can be used in the implementation of embedded real-time software. It is observed that, as an AOP language AspectC++ is nearly as powerful as its Java version AspectJ.

In the evaluation of the implemented Audio Switch project, two different evaluation approaches are applied. First, the two different implementations are considered from the point of software quality view. Then the project implementations are examined with respect to their embedded real-time performance.

The crosscutting concerns in the project are selected as to be the most common crosscutting concerns, which can be seen in nearly all software projects. So the evaluation results can be generalized to the field of embedded real-time systems.

The evaluation process is carried out by, gradually increasing the amount of crosscutting code in the system software. This is done to show the change in

the impact of AOP to embedded real-time systems with respect to the increasing amount of aspect code in the system software.

Looking at the software quality metric results given in the Evaluation chapter, it can be concluded that AOP provides an observable improvement in the software quality attributes. In other words AOP is said to improve the reusability, maintainability, testability and understandability of the system software. Moreover, from the point of programmers view AOP makes the system's crosscutting concerns more modular. So, programmer gains the ability to easily control and modify the crosscutting functionality of the system, as it is the case for functional-concerns in OOP.

Embedded real-time performance results, shows that, AOP provides a significant performance improvement in CPU usage, memory usage and run-time of the system software. When we consider the resource constraints and performance requirements of embedded real-time systems, it is a fact that AOP has a significant impact on embedded real-time systems. Especially, when we look at the difference between worst-case and average run-time results of the Audio Switch project AOP is said to prevent the unpredictable timings in message passing issues.

When we consider the results of the evaluation process as a whole, it can be said that using AOP techniques in the implementation of crosscutting concerns has a positive impact in the field of embedded real-time systems. This impact becomes more observable with the increasing amount of aspect code in the systems software.

Since the impact of AOP becomes more observable with the increasing amount of aspect code, as a future work implementation of an embedded project totally by using AOP techniques can be considered. Furthermore AOP can be examined as an alternative to OOP. Since OOP is not widely used in embedded real-time systems because of its performance overhead, AOP can be more suitable for the applications of this field.

REFERENCES

[1] S. Agrawal & P. Bhatt, "Real-time Embedded Software Systems", TATA Technology Review, 2001

[2] Nissanke, N., "Real-time Systems", Prentice Hall, 1997

[3] Burns, A. and A. Wellings, "Real-time Systems and Programming Languages", Addison-Wesley, 2001

[4] Wehrmeister, M.A., Pereira, C.E., Becker, L.B.; "Object-oriented methodology to the development of embedded real-time systems", 3rd IEEE International Conference on Industrial Informatics, 2005

[5] Hayes, R.G., “Real-time Java”, Department of Electrical Engineering and Computer Science College of Science and Engineering Loyola Marymount University: Los Angeles, USA, December 2000,

[6] Guillem Bernat,Alan Burns and Albert Llamosi, "Weakly Hard Real-Time Systems", IEEE TRANSACTIONS ON COMPUTERS, VOL. 50, NO. 4, APRIL 2001

[7] Joseph C. Sremack, "Investigating Real-Time System Forensics", Workshop of the 1st International Conference on Security and Privacy for Emerging Areas in Communication Networks", 2005.

[8] Tomoji Kishi, Natsuko Noda, "Aspect-Oriented Context Modeling for Embedded Systems", Early Aspects Workshop 2004

[9] Martin, R., Newkirk, J., and Koss, R., "Agile Software Development, Principles, Patterns, and Practices", Prentice Hall, 2003.

[10] Dean Wampler, "Aspect-Oriented Design Principles: Lessons from Object-Oriented Design", AOSD Workshop 2007

[11] Bruno Harbulot, "Introduction to Aspect-Oriented Software Development", ELF Developers' Forum – Manchester, October 2005

[12] Lars Rosenhainer, "Identifying Crosscutting Concerns in Requirements Specifications", Early Aspects Workshop, 2004

[13] Awais Rashid and Lynne Blair, "Aspect-oriented Programming and Separation of Crosscutting Concerns" British Computer Society The Computer Journal The Computer Journal, Vol. 46, No. 5, 2003

[14] Olaf Spinczyk , Daniel Lohmann a and Matthias Urban, "Advances in AOP with AspectC++", from

http://www.aspectc.org/fileadmin/publications/somet-2005.pdf

[15] Miller, S.K.; "Aspect-oriented programming takes aim at software complexity", IEEE Computer Journal Volume 34, Issue 4, April 2001

[16] Kiczales, G., et al. "Aspect-Oriented Programming.", ECOOP. June 1997.

[17] " Motorola MVME 5100 Data Sheet ", from https://mcg.motorola.com/us/ds/pdf/ds0008.pdf

[18] "VxWorks Center", from

http://www.windriver.com/vxworks/index.html

[19] Rosenberg, L.H., "Applying and Interpreting Object Oriented Metrics", Software Assurance Technology Center, July 2003, from

http://www.satc.gsfc.nasa.gov/

[20] Chidamber, S., Kemerer, C., "A metrics suite for object oriented design", IEEE Transactions on Software Engineering 20, Pages 476–493, 1994

[21] Shiu Lun Tsang; Clarke, S.; Baniassad, E., "An evaluation of aspect-oriented programming for Java-based real-time systems development", Proceedings of Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2004.

[22] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat, "Lean and Efficient System Software Product Lines - Where Aspects Beat Objects", Transaction of Aspect-Oriented Software Development (TAOSD), 2006

[23] Ian Sommerville, "Object-Oriented Design", March 2004 from

http://sunset.usc.edu/~neno/cs477_2003/March4.ppt

[24] Daniel Lohman, Olaf Spinczyk, "Aspect-Oriented Programming with C++ and AspectC++", 2005 from

http://www.aspectc.org/fileadmin/publications/aosd-2005-tut-2x2.pdf

[25] Harrison, R., Counsell S.J., and Nithi R.V. “An Evaluation of the MOOD Set of Object-Oriented Software Metrics” in Proceedings IEEE Transactions on Software Engineering, Vol.24, No.6, 1998, pp. 491-496.

[26] Baris Aydinoz, Semih Bilgen, " The effect of design patterns on object-oriented metrics and software error-proneness ", 2006, from

http://etd.lib.metu.edu.tr/upload/2/12607591/index.pdf

APPENDIX A

AspectC++ Language Quick Reference

a) Syntax Extensions

The AspectC++ syntax is an extension to the C++ syntax defined in the ISO/IEC 14882:1998(E) standard.

class-key:
  aspect

declaration:
  pointcut-declaration
  advice-declaration

member-declaration:
  pointcut-declaration
  advice-declaration

pointcut-declaration:
  pointcut *declaration*

pointcut-expression:
  constant-expression

advice-declaration:
  advice *pointcut-expression* **:** *order-declaration*
  advice *pointcut-expression* **:** *declaration*

order-declaration:
  order (*pointcut-expression-list*)

pointcut-expression-list:
  pointcut-expression
  pointcut-expression **,** *pointcut-expression-list*

b) Aspects

aspect *A* **{ ... };**
  defines the aspect *A*

aspect *A* : *public B* **{ ... };**
  A inherits from class or aspect *B*

c) Advice Declarations

advice *pointcut* : **before**(...) {...}
  the advice code is executed before the join points in the pointcut

advice *pointcut* : **after**(...) {...}
  the advice code is executed after the join points in the pointcut

advice *pointcut* : **around**(...) {...}
  the advice code is executed in place of the join points in the pointcut

advice *pointcut* : **order**(*high*, ...*low*);
  high and *low* are pointcuts, which describe sets of aspects. Aspects on the left side of the argument list always have a higher precedence than aspects on the right hand side at the join points, where the order declaration is applied.

If the advice is *not* recognized as being of a predefined kind (i.e. **before**, **after**, **around**, or **order**), it is regarded as an **introduction** of a new method, attribute, or type to all join points in the pointcut.

d) Match Expressions

Type Matching

`"int"`
  matches the C++ built-in scalar type `int`

`"% *"`
  matches any pointer type

Namespace and Class Matching

`"Chain"`
  matches the class, struct or union *Chain*

`"Memory%"`
  matches any class, struct or union whose name starts with "Memory"

Function Matching

`"void reset()"`
  matches the function *reset* having no parameters and returning `void`

`"% printf(...)"`
  matches the function *printf* having any number of parameters and returning any type

`"% ...::%(...)"`
  matches any function, operator function, or type conversion function (in any class or namespace)

`"% ...::Service::%(...) const"`
  matches any const member-function of the class `Service` defined in any scope

`"% ...::operator %(...)"`
  matches any type conversion function

Template Matching

`"std::set<...>"`
  matches all template instances of the class *std::set*

`"std::set<int>"`
  matches only the template instance *std::set<int>*

`"% ...::%<...>::%(...)"`
  matches any member function from any template class in any scope

e) Predefined Pointcut Functions

Functions

call(*pointcut*) $N \rightarrow C_C$
  provides all join points where a named entity in the *pointcut* is called.

execution(*pointcut*) $N \rightarrow C_E$
  provides all join points referring to the implementation of a named entity in the *pointcut*.

construction(*pointcut*) $N \rightarrow C_{Cons}$
  all join points where an instance of the given class(es) is constructed.

destruction(*pointcut*) $N \rightarrow C_{Des}$
  all join points where an instance of the given class(es) is destructed.

pointcut may contain function names or class names. A class name is equivalent to the names of all functions defined within its scope combined with the || operator (see below).

Control Flow

cflow(*pointcut*) $C \rightarrow C$
: captures join points occuring in the dynamic execution context of join points in the *pointcut*. The argument pointcut is forbidden to contain context variables or join points with runtime conditions (currently cflow, that, or target).

Types

base(*pointcut*) $N \rightarrow N_{C,F}$
: returns all base classes resp. redefined functions of classes in the *pointcut*

derived(*pointcut*) $N \rightarrow N_{C,F}$
: returns all classes in the *pointcut* and all classes derived from them resp. all redefined functions of derived classes

Context

that(*type pattern*) $N \rightarrow C$
: returns all join points where the current C++ `this` pointer refers to an object which is an instance of a type that is compatible to the type described by the *type pattern*

target(*type pattern*) $N \rightarrow C$
: returns all join points where the target object of a call is an instance of a type that is compatible to the type described by the *type pattern*

result(*type pattern*) $N \rightarrow C$
: returns all join points where the result object of a call/execution is an instance of a type described by the *type pattern*

args(*type pattern, ...*) $(N,...) \rightarrow C$
: a list of *type patterns* is used to provide all joinpoints with matching argument signatures

Instead of the *type pattern* it is possible here to pass the name of a **context variable** to which the context information is bound. In this case the type of the variable is used for the type matching.

Scope

within(*pointcut*) $N \rightarrow C$
: filters all join points that are within the functions or classes in the *pointcut*

Algebraic Operators

pointcut **&&** *pointcut* $(N,N) \rightarrow N,\ (C,C) \rightarrow C$
: intersection of the join points in the *pointcuts*

pointcut **||** *pointcut* $(N,N) \rightarrow N,\ (C,C) \rightarrow C$
: union of the join points in the *pointcuts*

! *pointcut* $N \rightarrow N,\ C \rightarrow C$
: exclusion of the join points in the *pointcut*

f) Join Point Types

Code

$C, C_C, C_E, C_{Cons}, C_{Des}$:
: any, *Call*, *Execution*, *Construction*, *Destruction*

Name

N, N_N, N_C, N_F, N_T :
: any, *Namespace*, *Class*, *Function*, *Type*

g) Join Point API

The JoinPoint-API is provided within every advice code body by the built-in object ***tjp*** of class ***JoinPoint***.

Compile-time Types and Constants

That [type]
: object type (object initiating a call)

Target [type]
: target object type (target object of a call)

Result [type]
: result type of the affected function

Arg::<i>::Type, *Arg::<i>::ReferredType* [type]
: type of the ith argument of the affected function (with $0 \leq i < ARGS$)

ARGS [const]
: number of arguments

JPID [const]
: unique numeric identifier for this join point

JPTYPE [const]
: numeric identifier describing the type of this join point (*AC::CALL* or *AC::EXECUTION*)

Runtime Functions and State

static const char ***signature**()
: gives a textual description of the join point (function name, class name, ...)

That ***that**()
: returns a pointer to the object initiating a call or 0 if it is a static method or a global function

Target ***target**()
: returns a pointer to the object that is the target of a call or 0 if it is a static method or a global function

Result ***result**()
: returns a typed pointer to the result value or 0 if the function has no result value

Arg::<i>::ReferredType ***arg**()
: returns a typed pointer to the argument value with compile-time index *number*

void ***arg**(*int number*)
: returns a pointer to the memory position holding the argument value with index *number*

void **proceed**()
: executes the original code in an around advice

AC::Action **&action**()
: returns the runtime action object containing the execution environment to execute (*trigger()*) the original code encapsulated by an around advice

Runtime Type Information

static AC::Type **type**()

static AC::Type **resulttype**()

static AC::Type **argtype**(*int* i)
: return a C++ ABI V3 conforming string representation of the signature / result type / argument type of the affected function

APPENDIX B

Motorola MVME 5100 Specifications

PROCESSOR

	MPC7410	MPC750	MPC755
Clock Frequency:	400 or 500 MHz	450 MHz	400 MHz
On-chip Cache (I/D):	32K/32K	32K/32K	32K/32K
Secondary Cache:	2MB	1MB	1MB

MAIN MEMORY

Type: PC100 ECC SDRAM with 100 MHz bus

Capacity: Up to 512MB on-board, expandable to 1GB with RAM500 memory mezzanines

Single Cycle Accesses: 10 Read/5 Write

Read Burst Mode: 7-1-1-1 idle; 2-1-1-1 aligned page hit

Write Burst Mode: 4-1-1-1 idle; 2-1-1-1 aligned page hit

Architecture: 64-bit, single interleave

FLASH MEMORY

Type: EEPROM, on-board programmable

Capacity: 1MB via two 32-pin PLCC/CLCC sockets; 16MB surface mount

Read Access (16MB port): 70 clocks (32-byte burst)

Read Access (1MB port): 262 clocks (32-byte burst)

NVRAM

Capacity: 32KB (4KB available for users)

Cell Storage Life: 50 years at 55° C

Cell Capacity Life: 5 years at 100% duty cycle, 25° C

Removable Battery: Yes

VMEBUS ANSI/VITA 1-1994 VME64 (IEEE STD 1014)

Controller: Tundra Universe

DTB Master: A16–A32; D08–D64, BLT

DTB Slave: A24–A32; D08–D64, BLT, UAT

Arbiter: RR/PRI

Interrupt Handler/Generator: IRQ 1–7/Any one of seven IRQs

System Controller: Yes, jumperable or auto detect

Location Monitor: Two, LMA32

COUNTERS/TIMERS

TOD Clock Device: M48T37V

Real-Time Timers/Counters: Four, 32-bit programmable

Watchdog Timer: Time-out generates reset

ETHERNET INTERFACE

Controller: Two Intel® 82559ER

Interface Speed: 10/100Mbps

PCI Local bus DMA: Yes, with PCI burst

Connector: One routed to front panel RJ-45, one routed to front panel RJ-45 or optionally routed to P2, RJ-45 on MVME761

ASYNCHRONOUS SERIAL PORTS

Controller: 16C550C UART

Number of Ports: Two, 16550 compatible

Configuration: EIA-574 DTE

Async Baud Rate, bps max.: 38.4K EIA-232, 115Kbps raw

Connector: One routed to front panel RJ-45, one on planar for development use

DUAL IEEE P1386.1 PCI MEZZANINE CARD SLOTS

Address/Data: A32/D32/D64, PMC PN1, PN2, PN3, PN4 connectors

PCI Bus Clock: 33 MHz

Signaling: 5V

Power: +3.3V, +5V, ±12V; 7.5 watts maximum per PMC slot

Module Types: Two single-wide or one double-wide, front panel or P2 I/O

PCI EXPANSION CONNECTOR

Address/Data: A32/D32/D64

PCI Bus Clock: 33 MHz

Signaling: 5V

Connector: 114-pin connector located on the planar of the MVME5100

POWER REQUIREMENTS

(not including power required by PMC or IMPC modules)

	+5 V ± 5%	+12 V ± 10%	–12 V ± 10%
MVME5100	3.0 A typ.	8.0 mA typ.	2.0 mA typ.

BOARD SIZE

Height: 233.4 mm (9.2 in.)

Depth: 160.0 mm (6.3 in.)

Front Panel Height: 261.8 mm (10.3 in.)

Width: 19.8 mm (0.8 in.)

Max. Component Height: 14.8 mm (0.58 in.)

PMC INTERFACE

Address/Data: A32/D32/D64, PMC PN1, PN2, PN3, PN4 connectors

PCI Bus Clock: 33 MHz

Signaling: 5V

Module Type: Basic, single-wide; P2 I/O

SCSI BUS

Controller: Symbios 53C895A

PCI Local Bus DMA: Yes, with PCI local bus burst

Asynchronous (8-bit mode): 5.0MB/s

Ultra SCSI: 20.0MB/s (8-bit mode), 40.0MB/s (16-bit mode)

Note: 16-bit SCSI operation precludes the use of some PMC slot 2 signals.

SYNCHRONOUS SERIAL PORTS

Controller: 85230/8536

Number of Ports: Two (IPMC761); one (IPMC712)

Configuration: IPMC761: TTL to P2 (both ports), SIM configurable on MVME761; IPMC712: EIA-232 to P2

Baud Rate, bps max.: 2.5M sync, 38.4K async

Oscillator Clock Rate (PCLK): 10 MHz/5 MHz

ASYNCHRONOUS SERIAL PORTS

Controller: 16C550 UART; 85230/8536

Number of Ports: Two (IPMC761); three (IPMC712)

Configuration: EIA-574 DTE (IPMC761); EIA-232 (IPMC712)

Async Baud Rate, bps max.: 38.4K EIA-232, 115Kbps raw

PARALLEL PORT

Controller: PC97307

Configuration: 8-bit bi-directional, full IEEE 1284 support; Centronics compatible (minus EPP and ECP on MVME712M)

Modes: Master only

POWER REQUIREMENTS

(Additional power load placed on MVME5100 series with IPMC installed)

	+5V ± 5%	+12V ± 10%	–12V ± 10%
MVME5100:	3.8 A max. 3.0 A typ.	8.0 mA typ.	2.0 mA typ
MVME5106:	3.8 A max. 2.6 A typ.	8.0 mA typ.	2.0 mA typ.
MVME5107:	4.7 A max. 3.5 A typ.	8.0 mA typ.	2.0 mA typ.
MVME5110-21xx:	3.8 A max. 3.1 A typ.	8.0 mA typ.	2.0 mA typ.
MVME5110-22xx:	4.7 A max. 3.5 A typ.	8.0 mA typ.	2.0 mA typ.

I/O CONNECTORS

	MVME761	MVME712M
Asynchronous Serial Ports:	Two, DB-9 labeled as COM1 and COM2	Three, DB-25 labeled Serial 1, Serial 2 and Serial 3
Synchronous Serial Ports:	Two, HD-26 labeled as Serial 3 and Serial 4 (user-configurable via installation of SIMs); two 60-pin connectors on MVME761 planar for installation of two SIMs	One, DB-25 labeled as Serial 4
Parallel Port:	HD-36, Centronics compatible	D-36, Centronics compatible
Ethernet:	10BaseT or 100BaseTX, RJ-45	Not available
SCSI:	8- or 16-bit, 50- or 68-pin connector via P2 adapter	8-bit, standard SCSI D-50

ENVIRONMENTAL

(Minimum of 400 LFM of forced air cooling is recommended for operation in the higher temperature ranges.)

	Operating	Non-operating
Commercial Temperature:	0° C to +55° C (inlet air temp. w/forced air cooling)	–40° C to +85° C
Extended Temperature:	–20° C to +71° C	–40° C to +85° C
Humidity (NC):	5% to 90%	5% to 90%
Vibration:	2 Gs RMS, 20–2000 Hz random	6 Gs RMS, 20–2000 Hz random

ELECTROMAGNETIC COMPATIBILITY (EMC)

Intended for use in systems meeting the following regulations:

U.S.: FCC Part 15, Subpart B, Class A (non-residential)

Canada: ICES-003, Class A (non-residential)

This product was tested in a representative system to the following standards:

CE Mark per European EMC Directive 89/336/EEC with Amendments; Emissions: EN55022 Class B; Immunity: EN55024

SAFETY

All printed wiring boards (PWBs) are manufactured with a flammability rating of 94V-0 by UL recognized manufacturers.

DEMONSTRATED MTBF

(based on a sample of eight boards in accelerated stress environment)

Mean: 190,509 hours

95% Confidence: 107,681 hours