Measuring the Perceived Overhead Imposed by Object-Oriented Programming in a Real-time Embedded System

Sumithra Bhakthavatsalam

Thesis submitted to the faculty of
Virginia Polytechnic Institute and State University
in partial fulfillment of requirements for the degree of

Master of Science
in
Computer Science and Applications

Dr Stephen H. Edwards, Chair
Dr James D. Arthur
Dr Dushan Boroyevich

16th May 2003
Blacksburg, Virginia

Keywords: Object-Oriented, Operating System, Embedded Software, Real-time, Kernel

Measuring the perceived overhead imposed by Object-Oriented programming in a Real-time Embedded system

Sumithra Bhakthavatsalam

(ABSTRACT)

This thesis presents the design and evaluation of an object-oriented (OO) operating system kernel for real-time embedded systems based on dataflow architecture. Dataflow is a software architecture that is well suited to applications that involve signal flows and value transformations. Typically, these systems comprise numerous processes with heavy inter-process communications. The dataflow style has been adopted for the control software for PEBB (Power Electronic Building Block) systems by the Center for Power Electronic Systems (CPES), Virginia Tech., which is involved in a research effort to modularize and standardize power electronic components. The goal of our research is to design and implement an efficient object-oriented kernel for the PEBB system and compare its performance vis-à-vis that of a non-OO kernel. It presents strategies for efficient OO design and a discussion of how OO performance issues can be ameliorated. We conclude the thesis with an evaluation of the advantages gained by using the OO paradigm both from the standpoint of the classically cited advantages of OO programming and other crucial aspects.

This work was supported primarily by the Office of Naval Research under Award Number N00015-01-1-0954 and the ERC Program of the National Science Foundation under Award Number EEC-9731677.

Acknowledgements

Thanks to my advisor Dr Stephen Edwards for his able guidance through my thesis and for providing financial support throughout the process.

I would also like to thank Dr James Arthur and Dr Dushan Boroyevich for their very valuable feedback on my work.

I am extremely grateful to Kuljeet Singh, who was involved in the development and evaluation of the non-OO version of this kernel, and Jinghong Guo, a member of the Center for Power Electronic Systems (CPES), Virginia Tech. for providing me support during my research.

Thanks to my family for their support and encouragement, without which none of this would have been possible.

Sumithra Bhakthavatsalam

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Efficiency is a major concern for real-time systems. Real-time software is often written in assembly or other low-level languages like C. Although OO programming offers advantages like modularity, code reusability and maintainability, programmers hesitate to use newer object-oriented techniques, as they are perceived to be less efficient. This thesis explores the object-oriented (OO) design techniques for embedded operating systems, taking into account real-time goals so that the design is efficient and meets system deadlines.

This embedded system kernel serves as a platform for power electronics control applications based on dataflow architecture. Dataflow, which supports asynchronous communication among components through communication channels, was chosen since it addresses the objectives of modularity, standardization and reusability of Power Electronics Building Block (PEBB) systems [25]. PEBB is a research effort to achieve plug-and-play power electronics components and overcome the disadvantages of the traditional approach to developing power electronics systems, which has been heavily driven by the hardware itself. The traditional approach causes lack of modularity, low flexibility, lack of standardization, and higher complexity, besides other disadvantages. The PEBB approach concentrates on modularizing and standardizing the power stage hardware and interfaces, while simultaneously modularizing and standardizing the digital control processing elements. This philosophy is applied also to the software running on embedded control processors.

Since the OO paradigm offers benefits including code reusability, better maintainability and enforced data encapsulation, and modularity, it is conceivable that the use of OO style for the PEBB embedded software could serve to address the aforementioned goals of PEBB systems. However, OO programming introduces efficiency issues that are a serious concern in the context of real-time systems where the goal is not just to produce the right results, but also to produce them at the right time.

The kernel described in this thesis is an OO version of an existing non-OO kernel implemented in C, called DARK (Dataflow Architecture Real-time Kernel). This OO kernel has been implemented in C++ and is consequently called DARK++.

1.1 Research Goals

The objective behind this research is to evaluate the extent to which object-oriented programming is suited to real-time embedded systems. The research goals are as follows:

- Implementing an efficient object-oriented operating system kernel for embedded power electronics control.
- Conducting performance experiments to compare its performance experiments to compare its performance with that of a non-OO kernel that is its counterpart.
- Drawing conclusions on how OO performance issues can be ameliorated by efficient design techniques.

1.2 Benefits from Object-Orientation

It is worth discussing the advantages that object-oriented programming offers that warrant its use. We discuss below, some important points in this regard:

- Compiler-enforced encapsulation in C++ offers more protection due to the notion of classes and different access levels for the various data members and behaviors - this applies to all OO systems in general, and therefore to our system as well.
- C++ offers more type safety with more static type checking – this is true of components that can be implemented by inheritance from a generic base class with parameterized sub-classes for different data types, as opposed to having a generic module that is used for all data types with appropriate typecasting as is done in non-OO implementations generally. Typecasting makes the system more error-prone.
- OO systems are more naturally extendible by inheritance and overriding of base class methods in subclasses.
- In our system, we found that with respect to the classically cited advantages of OO programming, viz., modularity, reusability and maintainability, our system is comparable to the non-OO system and does not offer any special gains because the non-OO system also has a well-defined structure with a standardized and reusable library of components.

1.3 Dataflow Architecture

The architecture of a system is defined in terms of its components and the interactions among them. Dataflow [20, 26, 27, 28] is an architectural style where the components of the system, which are concurrently executing processes, communicate asynchronously by the use of communication channels known as data channels. This architectural style is characterized by completely independent processes, with no shared memory, driven only by signals or data that are received through the data channels.

The control software for a PEBB system can be modeled on these lines because its operations are essentially based on signal flow and value transformations. A library of reusable components for constructing PEBB control applications has been proposed by Guo[20]. Guo calls these dataflow components Elementary Control Objects (ECOs).

Dataflow applications are typically depicted as dataflow graphs (DFGs) in which the components (ECOs) form nodes and the data channels form arcs.

1.4 An Object-Oriented Operating System

The OO paradigm in software design and development offers some significant advantages from a software engineering standpoint: modularity, code reusability and improved maintainability. There have been a number of operating systems that have been implemented using this paradigm. The world's first commercial object-oriented operating system is Genera, which is designed for a networked environment [14].

An object-oriented operating system may be defined as an operating system in which [6]:

- Resources, which may be hardware resources such as the CPU, memory segments or printers, or software resources such as files and programs, are available in the form of objects.
- Any program that runs on the operating system makes use of the provided classes/objects.
- All subsystems are provided as class structures.

DARK++ is an object-oriented operating system where the kernel is an object and all resources (data structures) that it manipulates or uses, such as the ready queue and the event queue are also objects. The processes scheduled and managed by the kernel are all objects embodying ECOs.

The system provides class structures for ECOs and the data channels that the applications running on DARK++ are to use.

1.5 Problem Statement

The DARK++ kernel has been designed and implemented as a platform to run PEBB systems, which are based on dataflow. The overall goal of PEBB-oriented research is to achieve modularization, standardization and reusability of power electronics components. The goal of DARK++ is to support this same modularity, standardization and reusability within the control software. Since the OO paradigm offers the advantages of software modularity and reusability, it is a natural fit with PEBB research. The DARK++ system comprises a library of ECOs that can be modified or extended very easily by using the classes and templates provided.

Due to the advantages of using OO programming for supporting PEBB systems, it is worth exploring its feasibility. Since we are dealing with real-time embedded systems, efficiency is a very important concern. We need to consider both the efficiency *requirements* in the given scenario, and also the efficiency *issues* that we need to overcome due to the use of OO programming.

Dataflow applications are comprised of processes that are essentially data driven. This necessitates causal ordering of the processes, which is done by the scheduler. However, for a real-time system, processes also have to be temporally ordered since every process has to meet its real-time deadline. In such a scenario, the overhead that the OS causes plays a crucial role. Moreover, dataflow applications typically comprise a larger number of smaller processes as a result of striving for modular, independent and reusable components. The natural consequence of a large number of processes is:

- high overhead in scheduling and context switching
- high frequency of inter-component communication

Over and above these efficiency issues, the OO paradigm introduces additional performance issues. The main sources for concern over extra run-time overhead are:

- Heap-allocated memory, due to the extensive dynamic creation of objects
- Dynamic binding, due to the use of virtual methods
- Method call overhead, due to a large number of small methods in classes

Each of these issues is discussed in detail in the following three subsections.

1.5.1 Heap-Allocated Memory

Many programmers perceive object-oriented systems as imposing more overhead due to the extensive use of dynamic memory allocation, dynamic creation and destruction of objects, and use of pointers or references. This overhead comes in the form of memory management and pointer dereferencing. In some systems, garbage collection is also a concern. Heap memory management in particular leads to unpredictable time delays in underlying system calls. In addition, *pointer chasing*, or following indirect references from one object to another as computation proceeds through a series of related objects, is a well-known problem in the context of OO systems. Pointer chasing increases memory traffic and decreases performance due to lack of data locality and cache inefficiency.

1.5.2 Dynamic Binding

Dynamic binding refers to deferring until run-time the association between a reference to a program operation (such as a class method) in calling code and the actual entry point of the corresponding segment of machine code that implements that operation. Typically dynamic binding in object-oriented systems occurs due to the use of virtual methods.

Virtual methods are operations that are declared in a base class, but which may have different implementations in subclasses. When other program code that is written to operate on any object—either an object of the base class or of any of its subclasses—makes method calls, the correct implementation of the method in question must be used. As a result, no fixed code address can be associated with that method call—a different address must be used depending on the particular (sub)class to which the object belongs.

Dynamic binding is usually implemented using indirection. The actual entry point is looked up at run-time in a dispatch table. The cost of performing the lookup and using the extra level of indirection to execute operations significantly increases the overhead of calling methods.

1.5.3 Method Calls

While dynamic binding can be avoided in many situations, method calls (calls to methods in general; these may not necessarily be virtual methods) still introduce performance concerns over the time spent in setting up the stack frame and saving registers. While this need not take any longer in an OO language than regular procedure or function calls in a non-OO language, OO design practices typically lead to more method calls. Because classes typically have a greater number of smaller methods, behaviors often use a greater number of method calls arranged in deeper call trees. This pattern leads to additional overhead in comparison to non-OO languages such as C.

It should be noted that having large methods to reduce the number of method calls would conflict with the goal of software reusability, because the more generic a component is, the more reusable it is. Thus, in general terms, methods in OO systems are small and generic, with behavior being extensible through inheritance. Subclasses usually have methods that perform a specific operation and make calls to more generic methods in the base class. Thus, the overhead of method calls is a potential performance concern in OO systems.

1.5.4 Summary

The high-performance requirement of real-time systems in general, and the specific requirements of dataflow applications, coupled with the OO performance issues discussed above, make the design and implementation of a high-speed OO kernel a challenge. DARK++ is a high-speed kernel that achieves its performance goals through careful design and use of certain strategies to ameliorate the mentioned performance issues. This thesis discusses the design of the kernel, the strategies for achieving efficiency, and the relative performance of DARK++ when compared to the non-OO DARK kernel.

Having achieved performance that is comparable to that of the non-OO implementation, we discuss the extent to which the often-stated advantages of OO programming (modularization, reusability and maintainability) have been realized in our system and compare the two implementations from these standpoints. Besides the classically cited advantages, there are other interesting benefits of using OO programming, such as compiler-enforced encapsulation and protection (since the class concept has various protection levels for its members), and more type safety. We consider the object-oriented design of our system and also some features designed by sacrificing the OO style and answer the important design versus performance tradeoff questions.

A discussion of all these issues helps us draw conclusions on the overall advantages gained by using OO programming in our system.

1.6 Organization of Thesis

This thesis is organized as follows. Chapter 2 provides some background information for the research and discusses related research work. It contains a discussion about dataflow architecture and its use in embedded systems. It also provides a brief insight into the design philosophies of a few existing object-oriented operating systems, and then discusses scheduling in the context of real-time systems. Finally, it covers some salient techniques on performance analysis of real-time systems. This chapter helps provide an insight into work done in the past and the similarities to our work. It also aids appreciating the unique aspects of our research and what distinguishes it from what has been discussed in the existing literature. Chapter 3 describes the OO design of the kernel. Here we first present the overall design with some discussion on the main base classes in the system and the API provided by DARK++. We further discuss the internals of DARK++, which covers the detailed design of the kernel. In Chapter 4, we provide the important API provided in the system. Chapter 5 discusses the implementation details of the important features in the DARK++ system and finally, in Chapter 6 we present the experimental evaluation of DARK++. This chapter contains experimental data of the performance of DARK++ and a comparison of its performance with that of the non-OO counterpart, DARK on three control applications – Open Loop Inverter, Closed Loop Inverter and Boost Rectifier, which are a representative set of applications with different numbers of ECOs and data channels. These are progressively more complex applications, with the first application containing 7 ECOs and 9 data channels, the second containing 9 ECOs and 20 data channels, and the third application being comprised of 18 ECOs and 31 data channels. We conclude the chapter with a discussion of the implications of the data gathered, the reasons for some performance overheads and explanations of the design techniques that contributed significantly to the performance of the system. Chapter 7 contains the conclusions of the research, its unique contributions and possible future work. An appendix containing the DARK++ source code is provided at the end of the thesis.

Chapter 2

Background Information and Related Research

This chapter is intended to provide the conceptual background of our work and to explore and discuss past research involving the key concepts employed in our research. The main areas from which this thesis could be viewed are – dataflow paradigm, object-oriented operating systems and real-time embedded kernels. Since our kernel supports embedded applications based on dataflow architecture, a discussion of dataflow architecture, the areas in which it has been employed earlier and its advantages and disadvantages in different contexts forms important background information.

We start with a discussion of dataflow architecture. This section discusses the most important differences between the dataflow paradigm and the von Neumann computational paradigm that make it advantageous to use the dataflow model in some contexts. We explore here the areas in which dataflow has been used widely in the past, followed by a discussion of its use of in PEBB systems. This is followed by a detailed discussion of the most important concepts involved in object-oriented operating systems, some specific examples of OO operating systems and the most important concurrency issues involved in OO operating systems. The section helps appreciate the aspects of OO operating systems that are pertinent to our kernel, and those that are not, and explains the nature of the issues that have to be dealt with in our system. We then move onto real-time systems and real-time kernels and a discussion of some of the commercial real-time operating systems to provide the reader insight into the key concepts involved in the design and implementation of real-time kernels. This will form a background to understanding what features our kernel does and does not provide vis-à-vis other existing real-time kernels. Finally, we also discuss the ideas involved in the performance analysis of our OO embedded kernel and comparison of its performance with that of the non-OO kernel.

2.1 Dataflow Architecture

A brief introduction to dataflow architecture has been given in Section 1.1. A dataflow program can be represented as a dataflow graph (DFG). Such a graph has vertices representing computation or comparison elements and edges representing the communication channels that interconnect these elements. Each element (vertex) in the graph can *fire* when there is data at each of its input ports. Dataflow programs can be constructed from simple computation elements with data communication among them. Dataflow allows for maximum concurrency because data flowing along different parts of the dataflow graph can be carried out in parallel. Dataflow graphs could be cyclic or acyclic. In cyclic dataflow graphs, nodes may fire more than once and so operands belonging to different executions are distinguished from each other using labels [33].

In a dataflow graph, a node or actor can have more than one way of being activated or fired; these are described by the input ports to that node that need to have data in them for the node to fire. This means that a node need not always have data in all of its input ports in order to be activated. It takes different actions based on which of its input data channels has/have data in them. In this way dataflow graphs are different from petrinets, in which nodes always need to have data in all of their input ports to fire.

It is important to note here that the dataflow architecture being discussed here is not the same as the concept of *data flow diagrams* (DFDs) in a software engineering context. DFDs are used to model the system when one begins design, after requirements elicitation. While DFDs can be used to produce an abstraction of the system at any level, the DFGs discussed here are more specific and they actually depict the processes in the system and the way they communicate.

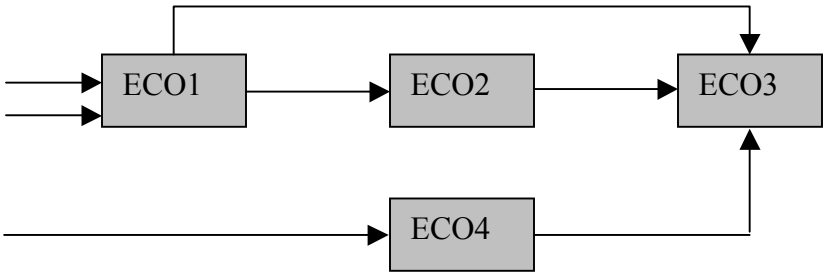


Figure 2.1. A Simple Dataflow Graph

The next section consists of a brief introduction to the dataflow computational model and discusses the model specifically in the context of multithreaded computing. In Section 2.1.2, we discuss the synchronous dataflow model, which is generally used in the design of real-time systems. Finally, in Section 2.1.3 we discuss the dataflow architectural model being applied to power electronics control applications.

2.1.1 Dataflow and Multithreaded Execution

The dataflow execution model is inherently different than the von Neumann stored-program execution model since it does not rely on the notion of a single point of control and that of a program counter. The strengths of this model complement those of the von Neumann model. The parallel nature of execution in the dataflow model helps overcome the two main problems of the von Neumann model, which are: *memory* latency, and *synchronization* overhead [15]. Latency is reduced due to the dynamic switching between ready computation threads. Dataflow also poses relatively low overhead in the distributed synchronization of hardware.

In computer architecture, two types of dataflow are considered: *static* and *dynamic*. In a static architecture, every edge of the dataflow graph can carry only a single data item at any time, thus restricting the number of instances of a process that can be active. In a

dynamic architecture, data items have tags associated with them so that edges can carry multiple data items at a time. Dynamic architectures allow more data parallelism.

The dataflow model and von Neumann serial control-flow model are generally viewed as two completely orthogonal execution models. However, starting with a pure dataflow graph, one can easily extend the model to support von Neumann style program execution. A region of nodes in a dataflow graph can be grouped together as a thread to be executed sequentially under its own private program control, while the activation and synchronization of threads are data driven. This allows for flexibility in combining dataflow and control flow evaluation. This new hybrid model also exposes parallelism at a desired level. A number of proposed hybrid multithreaded models have had their origins in either static dataflow or dynamic dataflow. The reader is advised to refer to the survey article by Dennis and Gao [16] for more details.

2.1.2 Synchronous Dataflow

Synchronous dataflow (SDF) is a decidable dataflow model; if we consider a particular topology of flows with the number of tokens produced and consumed by each node (actor) being fixed, it is decidable: whether a program will deadlock, and also whether a program has infinite execution that consumes bounded memory for storing pending tokens. In SDF, every node is characterized by the number of data items that it consumes and produces on each firing. The inputs and outputs of each node are labeled with an integer constant. For an input, this is the number of tokens required on that input stream in order to fire the actor and for the outputs, it is the number of tokens that are produced by a firing.

The decidability plays a useful role in certain types of real-time systems like some signal processing systems that involve repeated and infinite execution of a well-defined finite computation on an infinite stream of data. In such cases where the system has real-time constraints, it is important that the schedule of actor firings be predictable. It is also imperative that a program never deadlocks. Moreover, since these systems take the form of embedded systems, it is important to ensure that the total memory devoted to storing unprocessed tokens is bounded.

A deadlock occurs when all actors are starved. The problem of deadlocks is undecidable for general dataflow models. It is also undecidable whether a program has an infinite execution that consumes bounded memory for storing pending tokens [17,18]. One possible solution would be to use a subset of dataflow in which these questions are decidable.

As we know, in a dataflow graph an edge connecting two nodes represents the flow of tokens (data items) between them. If we assume that the topology of such flows is fixed, and that the number of tokens produced and consumed by each actor is fixed, then both deadlock and bounded memory are decidable. To appreciate this, we will briefly visit the topic of balance equations.

Suppose that an actor A is connected to an actor B. If A produces N tokens on an output and B requires M tokens on its input to fire, the number of times of firing of A and B are respectively f_A and f_B then the balance principle requires that [15]:

$$f_A N = f_B M$$

The balance principle is trivially satisfied if either $f_A = f_B = 0$ or $f_A = f_B = \infty$. However, if there is a bounded positive solution for f for every actor such that all balance equations (for all edges in the dataflow graph) are satisfied, then there may be a finite but non-zero set of firings that achieves balance. For a connected dataflow graph, if the balance equations have a non-trivial solution, then they have a unique smallest positive integer solution [19]. If they have no solution, then there is no bounded memory infinite execution. No non-trivial solution means that there is a simple finite algorithm that can determine whether the graph will deadlock. In this way, SDF makes both the bounded memory question and the deadlock question decidable, thus making them very useful for real-time embedded systems.

The SDF model however has some disadvantages. First, it comes at a very high cost in expressiveness. Since token production and consumption patterns are fixed, an application cannot use the flow of tokens to effect control; i.e., conditional variations in the flow are not allowed. Since signal processing applications too often involve a certain degree of conditional computation, the model is overly restrictive.

Our system provides different types of data channels for asynchronous and synchronous dataflow – the message queues, which are data channels that carry multiple data items can be used for asynchronous dataflow while the mailboxes, which are data channels of unit capacity can be used for synchronous dataflow.

2.1.3 Dataflow in Power Electronics Building Block (PEBB) Systems

The dataflow approach supports component-oriented reusability and parallelism. A relatively recent development has been to use dataflow in PEBB systems in which most computations can be expressed in terms of signal flows and value transformations as it is ideally suited to these kind of requirements. A power electronics control system could be seen as composed of many basic functional blocks or Elementary Control Objects (ECOs). Dataflow is used to organize these ECOs. ECO-Dataflow architecture is being used to synthesize modularized, standardized, reusable and automatic configurable power electronics systems [20].

Section 2.1.3.1 explains the concept of elementary control objects and Section 2.1.3.2 describes how dataflow architecture is used to combine these objects and build a control application.

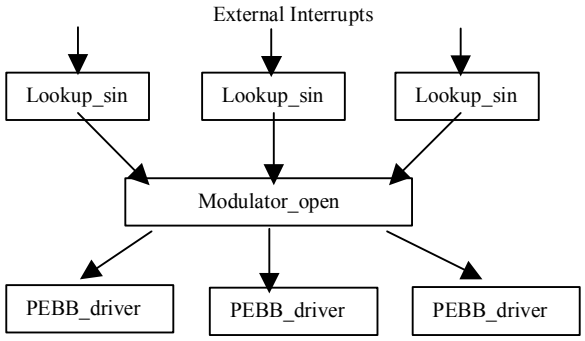


Figure 2.2 Open Loop Control for a 3-Phase Inverter

2.1.3.1 Elementary Control Objects (ECOs)

An ECO is defined as a *functionally self-contained, independent and concurrently executing* entity having *standard interface* and *implemented by multiple methods* [21].

An ECO interacts with the outer system using well-defined inputs and outputs. It has no knowledge of the other ECOs to which it is connected. Its function is to take the input, process it and output results if any. ECOs are naturally independent and can execute concurrently for distributed control of power electronics systems. A library of reusable ECOs for power electronics control applications has been proposed. These ECOs, when wired together, facilitate rapid development of power electronics control applications.

ECOs can also be considered as software components that replace the equivalent hardware. These are reusable, as a library of frequently used ECOs can be maintained. However, these are usually much slower than the equivalent hardware components.

ECOs operate on *firing rules*. A firing rule specifies the input channels on which an ECO should wait for inputs before starting execution.

2.1.3.2 ECO-Dataflow Architecture

The ECO-approach and dataflow can be combined to obtain a component-based software architecture. Here we treat ECOs as nodes or processes, which are connected to each other through data channels, which correspond to edges in the dataflow architecture. The ECO-Data Channel approach introduces modularity and reusability in the design of power electronics control software and also encourages development of control applications as a combination of self-contained modular components. The control algorithm is represented as a Dataflow graph composed of ECOs and data channels.

Figure 2.2 shows an example dataflow graph. This is for the open loop control of a three-phase inverter. This application consists of seven ECOs, and it looks up and derives the appropriate duty cycle commands to send to each phase leg of the power stage being

controlled. It should be noted here that every ECO in this dataflow graph has a specific pre-defined function and can be part of the ECO library maintained by the application programmer so that it can be reused later.

2.1.3.3 Why Dataflow?

Traditionally, embedded software is written in the "main-program-and-subroutine" style, in light of the requirement of highly optimized code. The major disadvantage of this is highly customized code that is hard to maintain/modify. Dataflow is a good alternative because it offers some distinct advantages [21]:

- Each component is an independent functional unit that need know nothing about its neighbors. This makes the components reusable since they are loosely coupled and can be employed in a variety of applications.
- Dataflow applications are naturally easy to reconfigure, since it only means addition/removal of nodes from the corresponding dataflow graph.
- Dataflow model is easy to map to distributed systems since all communication takes place only through data channels, which can be implemented as shared variables (for processes on the same processor) or as networked message transmission (for those on different processors).

The DARK++ system provides a library of reusable software components, which can be populated and used after a domain analysis of power electronics control problems to identify the recurring process tasks. The kernel runs dataflow applications, which consists of ECOs and data channels by scheduling and running processes (as threads), that represent the software of the ECO components.

2.2 Object-Oriented Operating Systems

Since our kernel is an object-oriented kernel, it is worth exploring some key concepts involved in the design and implementation of OO operating systems. This section deals with concepts and the following section explains briefly, some specific examples of OO operating systems.

Central to the discussion of object-oriented operating systems is an understanding of what distinguishes an *object-oriented system* from a merely *object-based* one. An *object-based* system provides for encapsulation and data hiding, and offers classes, modules or packages for the creation of object instances. On the other hand, an *object-oriented* system also provides class hierarchy by using inheritance.

Campbell, et al [2] classify OO operating systems as those that are merely designed and implemented in an OO style, and those that, in addition, provide services in an OO architecture; i.e., provide for the dynamic definition of class hierarchies for systems programs and applications. In the detailed discussion of the principles of design for the latter type of OO operating systems, they identify three main principles:

1. The hardware-system software interface of the system should be modeled in what they call a "framework". The state of the model reflects the state of the system and the system can be manipulated by modifying the framework. This requires constructing an object-oriented model of the hardware so that it can be mapped to classes.

2. Communication mechanism should be uniform for both application-to-system as well as application-to-application communication. This means that method invocation on objects should be uniform and independent of - whether an object is a local or a remote object, whether it resides in disk or memory or shares a virtual address space with the invoking object, and whether it is a system service object or an application object.

3. The operating system should form taxonomy for operating system algorithms and data structures so that it can be refined in future, through experimentation and evaluation. In other words, the OO operating system should be organized as a member of a family of operating systems.

These were the concepts underlying the design of the Choices operating system.

2.3 Some Object-Oriented Operating Systems

This section presents the discussion of the unique design features and concepts adopted in the design and implementation of a few existing OO operating systems. The OO operating systems discussed are Choices, Apertos, Merlin, Tornado and PEACE.

2.3.1 Choices

Choices is an OO operating system that is based on the concepts outlined in Section 2.2. It realizes the mentioned concepts by the use of *frameworks*, uniform communication mechanism and a mechanism for inter-address-space method invocation [1, 2]. Each of these is discussed in turn in the following subsections.

2.3.1.1 Frameworks

Choices designers have exploited class inheritance in OO programming to realize the concept of frameworks and to make the operating system algorithms and data structures independent of specific hardware. Abstract and concrete classes are used for this purpose. An abstract class is used to specify the most general protocol for any machine dependency, operating system mechanism / policy, or a design decision. A concrete class refines the implementation of an abstract class.

2.3.1.2 Uniform Communication Mechanism

Uniform access in Choices is achieved by performing all services by object method invocations. When the invoked object and its invoking object reside in the same address space, the mechanism is straightforward and is the procedure that occurs in regular C++ programs. However, inter-address-space method invocation is a special case where we need a specific mechanism that achieves access with the difference in the address spaces of the two objects being transparent to the client object. We discuss below, the inter-address-space method invocation in Choices.

2.3.1.3 Inter-Address-Space Method Invocation in Choices

An application running on Choices uses the class, ObjectProxy to invoke a method on an object resident in a different address space than its own. This is brought about by the use of method invocation in C++. Every object in C++ has a pointer to a table containing the addresses of the entry points of each of the methods of that object. This address is looked up from the table when a method is invoked in run-time, by using an offset determined during compilation. For virtual functions, the table entries in a subclass are replaced by the addresses of the redefined methods in the subclass. This is used in the inter-address-space method invocation in Choices. ObjectProxies act as surrogates to other objects. However, this process is not visible to the client object requesting a method invocation. The method invocation on ObjectProxy in turn invokes a method lookup function, which performs the actual inter-address-space method invocation by using the information provided by the nameserver and the descriptor for the target object, which is stored in ObjectProxy.

There are three main methods of inter-address-space method invocations, which are: shared memory, distributed virtual memory and message passing.

The *Shared Memory* concept is used when an application and a server object reside on the same machine. In this case, the kernel lookup function adds the requested object's data and code to the virtual address space of the application, for the duration of the call.

Distributed Virtual Memory is used when the objects reside on different machines of the same architecture. For this, one of two options is used. If the remote object is small or if it is large but sparsely accessed, then the remote object can be made to share the invoking object's data via network, to make it appear as if it were local. However, if the remote object is too large, and is accessed frequently, then the process is transferred from its CPU ProcessContainer on the local machine to that on the remote machine.

Message Passing and RPC is used when the two objects reside on different machines of different architectures. In this case, the arguments are copied between the two machines. Type information is used to encode the arguments. Stubs or wrappers are used for this. Stubs are used to encode the arguments in the messages to the remote machine. A stub at

the other end is then used to decode the arguments and invoke the method on the remote object.

2.3.1.4 Taxonomy

Structuring Choices as a class hierarchy of operating systems algorithms and data structures helps in the design of a family of operating systems that can be later customized for particular hardware architectures and applications. Abstract classes are used to define interfaces to the components and concrete classes are used to provide a particular customization.

2.3.2 Apertos

Apertos [6] introduces the notion of *reflectors* or *metaspaces*. Metaspaces hold definitions of Apertos objects' behaviors. A metaspace can hold any number of objects. To change the behavior of an object, it should be moved from its current metaspace to a different one. Every simple object is managed by a meta-object, which holds a reference to the corresponding metaspace.

Invoking an object is done by calling into its meta-space. In this way, the target object is located and the method dispatched. If computation is to be done at the meta level rather than at the base level, the target for the invocation is a meta-object instead of the base object.

2.3.3 Tornado

Tornado [6] uses an independent object to represent every virtual and physical resource so that the independence of all resources is ensured.

There are three main special features of Tornado:

- *Clustered Objects:* An object can be partitioned into representative objects so that independent requests on different processors are handled by different representatives of the object. This helps in efficient handling of simultaneous requests to a single resource.

- *Protected Procedure Call:* This facility uses the locality and concurrency of client requests for an object, to determine the way the requests are serviced. Repeated requests to the same object are serviced on the same processor and concurrent requests are processed by different server threads without the need for data sharing and synchronization.

- *Semi-automatic Garbage collection scheme:* Facilitates localizing lock accesses. All locks are internal to objects they are protecting. No global locks are used. [6]

2.3.4 PEACE Operating System and the 'Naming' Concept

PEACE is an OO operating system for massively parallel systems [3].

As the number of nodes in massively parallel systems with distributed memory architecture increases, bootstrapping the system becomes a challenge and an important task since supplying each of the nodes with an operating system could take an unduly long time. Usually, during startup of a system, neither all of the operating system services, nor all of the nodes in the system are needed. Hence, a wise method of starting up the system is to start up with only those operating system services that are really necessary at just one, or a few nodes. Further services are loaded on demand, and the further nodes are *incrementally bootstrapped* on demand. This helps in dynamically scaling up a parallel system.

In PEACE, the concept discussed above is realized by introducing what is known as a *dual object*. This is based on the fact that an object in an operating system service manager has two types of data – *public data*, which may be accessed commonly, and *private data*, which are used internally, only by the manager. Both these parts are held together in a prototype by the manager. The public part only is held in a *likeness* by the client. When a dual object related call is instantiated, the likeness is sent to the prototype's site, along with the required destination address. At the prototype's site, the public and the private data are reunited before the actual call is performed. This is done by a process called *client*.

Naming is done by using *nameservers*. Every service providing thread registers a symbolic name and its address in the name space, by a process called *Name Export*. The client looks the name space during object construction, to determine the address of a thread. This is called *Name Import*. Names within a single name server are unique. If a name has to appear multiple times, then the name space is partitioned into different name servers. A name can then appear in two different name servers.

2.3.4.1 Incremental Loading

In PEACE, loading of a service happens on demand and this is termed *incremental loading*. When a client looks up the name server and finds a name resolution failure, a *server fault* has occurred. This fault is handled by loading the missing object. Server fault handling is done by the *Entity Server*. The entity server handles not only system level services, but also application services.

A *nameserver fault* is raised when the server detects the absence of the name server specified as its export specification. Nameserver faults are handled by a separate system called *Name Usher*.

2.3.4.2 Naming Strategies

A client consults the name sever with a name either to add it up, if it is an Export, or to determine an address, if it is an Import. The implications of finding / not finding the name are different in both the cases. This gives rise to four possibilities.

"A behavior depending on the existence of a name is classified as a strategy." [3]
When a name is found in the name server during an Import, then a *contact mine* is said to have been established. The concept of contact mine is realized in PEACE by using: inheritance in the object-oriented design, and the concept of dual objects.

If functions in a base class are redefined in its derived classes, the call of such a function in a base class object causes ambiguity about which of the redefinitions (i.e., which subclass) is to be considered. Hence, only the base class methods are considered in this case. However, if one requires using methods that are level-specific, then virtual functions should be used, in which case resolution will happen dynamically (at run-time).

By using the concept of dual object, the name server uses the contact mine name and the object on which an import operation is defined.

2.3.5 Summary and Discussion

The most important aspects of OO operating systems as can be gathered from the above discussion are the taxonomy (or framework of classes) provided by the system, and communication mechanisms between objects, which subsumes handling method calls between objects residing in different address spaces and between objects residing in a distributed setup. In Choices, we saw the concept of frameworks with abstract and concrete classes defined for generic and hardware-specific code, respectively. DARK++ is a kernel meant for power electronic embedded systems. Since in embedded systems, the kernel essentially represents the entire operating system, there are no device drivers that may be hardware specific. This is the case with DARK++. Here the scheduler and its internal data structures are independent of the underlying hardware too. The system contains the abstract class defined for the ECOs in general, and concrete classes representing the actual ECOs enable having a library of reusable, modifiable ECO components. This is a *framework* that the system provides, although not necessarily in the Choices sense of demarcating the hardware-specific parts from the generic parts of the code. However, there are some parts of the kernel – the code for context switching and that for interrupt handling, which are written in assembly, that are hardware specific. In fact the high-speed context switching implemented as part of the kernel is based on the dual-register-set hardware provided by ADSP 21xxx on which our kernel runs. However,

it is natural to have customized parts of the code specific to hardware, since the kernel is meant for an embedded system. Moreover, these are not part of the OO design and are independent of the remaining class framework.

Processes in DARK++ run as threads in their individually allocated "stack spaces", which in reality, reside on the heap. For a distributed setup, we only need to add a protocol to the kernel, which transparently composes network messages for data to be sent from a process residing on one processor to one residing on another.

2.4 Concurrency in Object-Oriented Operating Systems

The objects in an object-oriented concurrent system are controlled by synchronization constraints. Synchronization constraints are specified on a per object basis and help to maintain data consistency. These constraints generally specify the conditions under which an object's methods may be invoked or enabled.

About specifying synchronization constraints for classes, Frolund [4] states that it is desirable to inherit synchronization constraints of a superclass in a subclass, whenever possible. The synchronization constraints can then be *incrementally modified* in the subclass, so as to make the conditions more comprehensive, or to add more conditions. Inheritance should be used for making synchronization constraints more restrictive at each level. This should be provided by incremental modifications.

Inheritance has the property of *factorization*, which refers to the fact that the properties of the superclasses also hold good for subclasses. Hence, synchronization constraints should be specified in such a way that the property resulting from them would apply to all the subclasses that will be derived from the class in which they are described. However, if synchronization constraints are specified as those that enable methods, then they may not necessarily hold good in the subclasses. Hence it is more desirable to specify synchronization constraints that disable a method, rather than enable it. This way, as inheritance happens, further subclasses keep getting more and more restrictive. This is because methods that are disabled in a superclass will always be disabled in a subclass. E.g. [4]: If a resource administrator is implemented as a class, resource-administrator with two methods *access* and *free*, and the synchronization constraint is that the resource administrator can grant access to resources only if the number of resources given out is greater than or equal to the maximum number of resources. The subclasses for the different resource administrators can have more restrictive synchronization constraints, that distinguish between heavyweight and lightweight resources and grant access based on a maximum number of resources, which is different for heavyweight and lightweight resources.

When synchronization constraints are specified in the form of restrictions, it is possible to specify a restriction as an aggregate of other restrictions. This helps in reuse by composition.

2.4.1 ACT++

ACT++ [5] is a class library in C++ that can be used for concurrent programming. It is based on what is known as the *Actor* model. In this model, programs are considered to consist of a number of active objects, called *actors*, which execute concurrently and communicate with each other by sending messages and receiving replies. An actor receives a request message from another actor, for the execution of a particular method. An object called *behavior* is responsible for the execution of this requested method. ACT++ overcomes the problems that manifest in concurrent programming with C++. Actors execute concurrently using independent threads. One of the most important features of ACT++ is that it overcomes the problem of Inheritance Anomaly in a concurrent environment. This is done by introducing the notion of *behavior sets*, *next-behavior-set function* and *object-state functions*.

"A behavior set is an object used by a behavior to process messages selectively depending on the behavior's state" [5]. This means that whenever there is a request for a method to be executed, the state of the behavior is first checked, and only if it is present in the *behavior set*, the method is executed.

The *object-state function* is a method present in the behavior that can be used to determine the current state of the behavior.

The *next-behavior-state function* uses the object-state function to determine which of the behavior sets is to be the current behavior set.

E.g.: Consider the bounded buffer problem. Since an "in" operation cannot be carried out on a full buffer, and an "out" operation cannot be carried out on an empty buffer, there will be three behavior sets corresponding to empty, partial and full states of the buffer. These three correspondingly would have the {in}, {in, out} and {out} functions as valid in their states. The three object-state functions are: empty(), partial() and full().

An important feature of ACT++ is the Cbox. This is an object that is used when an actor expects some response while making a request. It is one of three types: FIRST, which stores only the first reply message, LAST which stores only the last reply message, and QUEUE, which enqueues all the reply messages.

ACT++ also handles concurrent I/O. It makes use of Interface Actors (IAs), Rboxes, and Wboxes. An IA is a special type of actor that manages I/O to a single file, using a file descriptor. It serializes I/O requests to a particular file, and can handle both synchronous and asynchronous I/O. Rboxes and Wboxes are character buffers that are used to read and write data from / to a file, respectively. Both Rbox and Wbox classes have the Wait method. Before requesting a Read operation, a behavior creates an Rbox object. When the data is requested, the Wait method is issued on the Rbox. If the Rbox has not yet received the data, the operation is blocked. Otherwise, it is continued. The Wait method in Wbox is used to check whether the Write operation has finished. If it has not, the

operation is blocked. Else, the call returns immediately. In both Rbox and Wbox, the return from blocked to unblocked state of the operation is automatic.

2.4.2 Concurrency Issues in Our System

Having discussed the main concurrency issues that OO operating systems in general, are often required to deal with, we discuss the idea of concurrency control in DARK++. In our system, threads have no issues of shared memory due to the unique aspects of the dataflow model. Threads communicate only by means of data channels. A thread awaiting data on its input port begins executing only when the source thread has finished writing data to it.

The only issue is when an interrupt occurs when a process is executing and attempts to say, write data to a data channel that was being read by the executing process thread. In these cases, the interrupt handler logs an event code associated with the interrupt and returns control back to the executing thread, so that the process that was executing when the interrupt occurred can finish execution and the interrupt event can be carried out when control returns to the scheduler thread. Hence the normal concurrency issues do not arise in DARK++.

The following section reviews the main concepts of real-time systems with the idea of providing some background information to understand the requirements of such a system that have to be supported by a kernel running on such a system.

2.5 Real-Time Systems

Real-time systems may be *hard real-time* or *soft real-time*. In a *hard real-time system*, failure to meet any of the deadlines can be catastrophic. Such a system has a very low tolerance to missed timing deadlines. E.g.: Automotive Engine Unit. In contrast to such a system, a *soft real-time system* is more tolerant to missed timing deadlines and missing is not catastrophic; for instance, in cell phones, occasional missing of deadlines lead to glitches and a poor quality of the voice transmitted but this may be tolerated, although undesirable.

In today's world, real-time systems are moving from the centralized to distributed control. In embedded controllers of consumer electronic products, it is cheaper to add different special purpose processors rather than to connect all devices to a single big processor. For the distributed nature of real-time systems, object-oriented design is well suited because of the modularity of design enforced by object-orientation. Further, it enables code reuse, and handles distributed objects in an elegant way. However, to be able to meet real-time constraints, the traditional object model needs to be enhanced to support two features [8]:

1. Timing constraints have to be incorporated along with functional specification of objects.

2. Concurrency control has to be handled.

The DIRECT [8] project group proposed the use of dynamic scheduling to integrate new components in the system, to facilitate response to unforeseen events, and continuous monitoring to check compliance of the system behavior with the specified constraints.

Following are some of the concepts employed in the DIRECT architecture:

- *Fine-grained global time base:* For a real-time system, the causal ordering of events is not sufficient. What is required is the temporal ordering. Further, the granularity of the time base should be fine enough to be able to distinguish the occurrence of events in the distributed system annotated by their timestamps.

- *Constraint Checker:* To support the specification of real-time behavior of objects, besides their functional behavior, timing annotations are required, either for the classes or for individual objects. This introduces the notion of *Timed Objects*, which have not only data abstraction, but also timing abstraction. Timing constraints are specified at the object interface and if an object violates its timing specification, then the system may report this. Application code can be automatically generated based on timing annotations.

- *Adaptive Scheduler:* This component determines the importance of tasks dynamically. Since the importance of a task may change in dynamic systems, this has to be handled by a scheduler.

2.6 Real-time Kernels

An operating system may be viewed as an organized collection of software extensions of hardware consisting of control routines for operating a computer and for providing an environment for execution of programs [29]. In an operating system, we may identify three specific task management functions. These are *task scheduling*, *task dispatching* and *intertask communication*. A *kernel* or *nucleus* is the smallest part of an operating system that performs these three functions.

There are different variants of the definition of a “kernel”. These are shown in increasing order of complexity and functionality from the center outward in the following diagram (Figure 2.3). Their functionalities are also shown in the diagram.

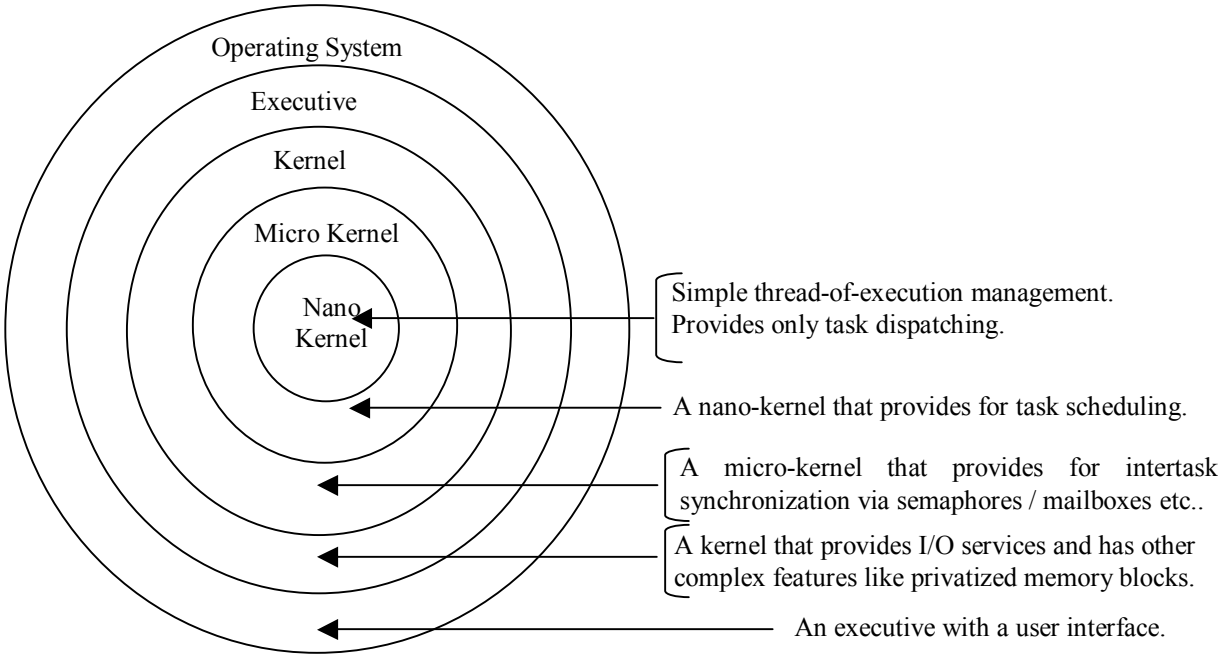


Figure 2.3 Hierarchy of different types of Kernels

The figure can be considered to be a hierarchy with the portion represented by every enclosing circle being a functional superset of that being represented by a circle within.

Since real-time systems generally have a minimalist design, in embedded real-time systems, the kernel represents the entire operating system. DARK++ is kernel since it provides task scheduling by maintaining an internal priority queue (the ready queue), task dispatching (runs the process thread that is at the top of the ready queue), and inter-task communication via data channels. Since it is a multithreaded system with no services for protected privatized memory blocks for various executing processes, it does not qualify as an executive.

2.6.1 Polled Loop Systems

Polled loop systems are the simplest form of real-time kernel. Polling is the testing of a particular condition repetitively until the condition succeeds. The system continues polling as long as an event does not occur. Since there exists only one executing task at any point of time, no intertask synchronization is required.

Polled loop systems with interrupts are a variant of polled loop systems, where the system waits a preset period after the test condition succeeds and then resets the flag corresponding to that condition. This is to avoid interpreting spurious events such as settling oscillations while a switch changes from OFF to ON condition, as events. The

required delay is created by using either hardware or software timer, and an interrupt to signal the end of the countdown period.

2.6.2 Phase-Driven or State-Driven Systems

A *phase-driven* or *state-driven* system is one in which the code is broken into fragments by the use of conditional statements or a finite state automaton (FSA). Only some process, like the compilation process, which are characterized by distinct phases that can be suspended temporarily, lend themselves well to the FSA model. The separation of processes allows multitasking by the use of coroutines, which will be discussed on the following subsection or other mechanisms.

Coroutines are used in conjunction with phase-driven code. These systems comprise processes that are coded in the phase-driven style and a central dispatcher. After every phase of a process is executed, a call is made to the central dispatcher. The dispatcher holds the program counter for the list of processes and selects the next process to execute in a round-robin fashion. The selected process executes its next phase and then returns to the central dispatcher.

2.6.3 Interrupt-Driven Systems

In interrupt driven systems, all tasks are scheduled by hardware or software interrupts and dispatched by the interrupt service routine.

Interrupts may be periodic or aperiodic. Based on this, interrupt driven systems may be classified as *sporadic* and *fixed-rate* systems, respectively. A system could also be *hybrid* if it has interrupts of both natures occurring.

Scheduling through hardware interrupts uses a clock or an external device that issues interrupts. These interrupts are directed to the interrupt controller, which further issues signals based on the order of arrival of the interrupts and their priority. Dispatching can also be handled through hardware if the computer architecture allows multiple interrupts. However, if only one interrupt level is permitted, then the interrupts service routine serves as the dispatcher. It reads the interrupt vector to determine which interrupt occurred and dispatches the appropriate task.

Context saving is necessary in these systems since the state of the machine has to be restored once the interrupt process is resumed. Generally, a *stack model* is used for context saving, especially in embedded systems where the number of tasks is fixed.

2.6.4 Foreground/Background Systems

Most embedded applications are foreground/background systems. They comprise a set of interrupt-driven or real-time processes that form the *foreground* and a set of non-interrupt-driven processes that form the *background*. The foreground tasks are either run as preemptive tasks with different priority levels or run round-robin. The background task can always be preempted by any of the foreground task; the background task, in this sense, is the lowest priority task.

The real-time systems discussed in section 2.6.3 may be viewed as specific cases of foreground/background systems. A polled loop system has the polled loop running as the background task. Making a polled loop systems interrupt-driven makes it a full foreground/background system. Phase-driven systems have no foreground tasks, and the phase-driven code runs as the background. Coroutines form background tasks. Interrupt-only systems are foreground/background systems that do not have a background.

Since any real-time system can be treated as a foreground/background system, foreground/background systems have been studied extensively.

2.6.5 Full-featured Real-time Operating Systems

Foreground/background systems can be enhanced into full-featured real-time operating systems by adding device drivers, network interfaces etc. These systems typically have a dynamic number of tasks and hence use the *task-control block model*, which is discussed in the following subsection.

2.6.5.1 Task Control Block Model

In this model, we associate with every task, a task control block. A task control block consists the context of task – program counter and registers, an identification string, a status and a priority, if applicable.

Task management is performed by keeping track of the state of each task, which at any point in time could be one of:

- Executing
- Ready
- Suspended
- Dormant

The *executing* task is the one that is currently running. Once the task is completed, it goes to the *suspended* state. An executing task enters the *ready* state either if its time slice expired, or if it was preempted. A suspended task enters the *ready* state if an event initiates it, and a dormant task becomes ready upon creation. Tasks that are waiting for a resource and are therefore not ready are in the suspended state. The *dormant* state is used

only in systems with a fixed number of task control blocks to determine the memory requirements beforehand.

The operating system checks the ready queue and executes the first ready task, thus changing the state of the task to *executing*. The task control block for this task is moved to the end of the queue and the task is removed from the ready queue. In addition to this, the suspended queue is also checked and resources that are freed are provided to tasks that have blocked waiting for them so that these tasks can become *ready*.

DARK++ is based on the task control block model, it is not a full-featured RTOS in the sense of having device drivers, network interfaces and the like. Rather, it is a real-time *kernel* that performs task scheduling, task dispatching and inter-task communications.

2.7 Priority-based Scheduling for Real-Time Embedded Systems

This section discusses some crucial and often-encountered aspects of scheduling in real-time embedded systems.

Priority-based Scheduling is a better way of constructing a hard real-time system than static scheduling. This is because it allows the operating system to execute tasks at just the required rates and not at higher rates. Moreover, urgent events are handled by high priority tasks that are made ready only when the corresponding events occur. As per Deadline Monotonic Analysis (DMA), assigning priorities monotonically with deadline ensures meeting all deadlines. This means the task with the shortest deadline is assigned the highest priority.

2.7.1 Priority Inversion, Priority Inheritance, and the Priority Ceiling Protocol

There is a problem of Priority Inversion in a priority-based real-time system. Consider a low-priority task, L running. It locks certain data for exclusive access, by using a semaphore, S1. Now, as L is running, a medium-priority task M is ready and so, the scheduler suspends L and starts executing M. As M is half way through its execution, a high-priority task, H becomes ready. Now H requires the data that has been locked by S1. But since L has a hold on this data, it must first be executed so that H can then gain a hold on it. But before L, M has to finish executing since it has a higher priority. Hence, now the scheduler suspends H and starts executing M. After M finishes executing, L is executed, and then H is resumed. This effectively means that there has been a *priority inversion* between M and H, since M ends up being executed before H. To overcome this problem, the concept of *priority inheritance* was introduced. This means, that when H is executing, if it requires the data on which L has a hold, then L is made to inherit the priority of H. So this makes it higher in priority than M, and would cause the scheduler to schedule L first, and then H, followed by M. This ensures that the relative priorities of M and H are unchanged and execute in the order- H, and then M.

However, the problem with priority inheritance is that it cannot guarantee a deadlock-free system. Consider two tasks, L and H that share two data buffers guarded by semaphores, S1 and S2. L might lock data guarded by S1 while it is executing, and may require to lock that guarded by S2 as well. However, before it does this, if H is ready, then the scheduler will cause L to suspend and H will start executing. Now if H also requires both data buffers guarded by S1 as well as S2, and has been able to gain access only to that corresponding to S2, but not to that corresponding to S1 (since it is already being held by task L), then there is a deadlock. To solve this problem, the *Priority Ceiling Protocol* was introduced. As per this protocol, each semaphore has a ceiling priority, which is the priority of the highest-priority task that can lock it. A task can hold several semaphores at a time. Then the Instance Inheritance algorithm is used, according to which, as soon as a semaphore is locked, the locking task raises its priority to the ceiling priority of the semaphore. When the semaphore is unlocked, the task's original priority is restored. Considering the deadlock scenario explained above, when L is executing, and has locked S1, its priority is raised to high since H can also lock the same semaphore (and so the ceiling priority of the semaphore, S1 is high). Since its priority is raised to high, when H is ready, L will not be suspended, and will continue to be executed. As soon as L releases S1, the priority of L is again reverted to low, and now task H is free to use S1.

The unique nature of dataflow applications obviates the priority-based scheduling issues and complexities discussed above. The topology of a dataflow application implicitly decides the priorities of the processes, at least partially. This is because, processes that depend on other processes for data in their input data channels, automatically have lower priority than the source processes. However, priorities are useful in resolving the order of execution of two processes that are both ready (have data in their input data channels) at the same time. Hence scheduling in DARK++ is based on the concept of firing rules, which has been mentioned briefly in Section 2.1.3.1.

2.7.2 OS Overhead: Scheduling and Switching

The switching costs of the OS can be modeled by adding the *switch-in* and *switch-out* costs [7], as shown:

```
Scheduler runs
  Begin Context Switch
    Run Task                      ⟶ Switch-in

    Re-queue task and switch      ⟶ Switch-out
```

Switching costs are an important metric for measuring the performance of a scheduler. Especially in the context of dataflow applications where the number of processes is large, switching times play a very crucial role since more processes implies more context switches. There are two types of context switches – application-application and

application-kernel. It is important to analyze the frequency of the two types of context switching and how each of them can be minimized. The high-speed context switching implemented in DARK++ is described in Chapter 3.

2.8 Survey of some commercial Real-time Kernels

2.8.1 MicroC/OS-II

MicroC/OS-II [22] is a fully preemptive kernel written in ANSI C. It can handle up to 64 tasks. In the current version eight are reserved for system use and the remaining tasks are for user applications. There are 64 priority levels for the tasks and each task has a unique priority assigned to it. These priorities are dynamic.

MicroC/OS-II is a scalable kernel. The user can select only the kernel features required in the application to reduce the amount of memory needed by the system. The kernel provides API for enabling and disabling the scheduler. When the scheduler is disabled, the kernel behaves as a non-preemptive system. There is also a facility using which one can find the CPU usage percentage for any application.

Because of its simplicity and lightweight, MicroC/OS-II is suitable for small embedded systems that require high-performance. However, there is a disadvantage - there are only 54 user tasks available and all the tasks should have different priorities. Also, it does not support POSIX standard unlike many other commercial kernels.

2.8.2 Analog Devices-VDK++

Analog Devices-VDK++ [23] is a preemptive multitasking kernel written in C shipped by Analog Devices along with its VisualDSP++ Integrated Development Environment (IDE). The threads in VDK++ can be written in C, C++ or assembly. Each thread in VDK++ is assigned a dynamic priority. The kernel supports either fourteen or thirty priority levels, depending on the processor's architecture. VDK++ provides two levels of protection for the code that needs to execute sequentially -*unscheduled regions* and *critical regions*. While an unscheduled region disables scheduling, a critical region disables both scheduling and interrupts and provides full protection to shared data. The scheduler in VDK++ is invoked whenever the kernel API called from either a thread or an ISR changes the highest priority thread.

VDK++ provides users various scheduling policy options. It supports cooperative scheduling and round robin scheduling in addition to preemptive scheduling. Although VDK++ provides services like events and semaphores, it does not provide support for communicating data between the threads. The object-oriented features of VDK++ cause a performance overhead, thereby making it unsuitable for high-performance applications.

2.8.3 VSPWorks

VSPWorks [24] is based on VxWorks, but unlike VxWorks, it is designed specifically for DSP-based systems. It provides preemptive multitasking and high-speed interrupt support on a range of DSP and ASIC core processors. VSPWorks has a single-processor (VSP) model. Here data objects and tasks can be moved from processor to processor transparently, with the operating system handling all the underlying inter-processor communication.

VSPWorks is a modular and scalable operating system. At compile time, the system definition tools automatically strip out all unused parts. The kernel also provides a suite of graphical tools to simplify and accelerate single- or multiprocessor application development.

VSPWorks follows a multi-layered design for abstraction and portability. The system has a highly optimized nanokernel that can manage a range of processes. Below this are the interrupt service routines (ISRs) for high speed interrupt handling. The microkernel sits above the nanokernel and handles preemptive multitasking of C/C++ tasks.

2.9 Performance Analysis of Real-Time Embedded System

This section discusses some important points with regard to performance analysis of real-time embedded systems. It further discusses the main techniques adopted for the same. While performance data can be gathered by the use of the profiler that is available with simulators etc., analysis is done manually by adopting one of the techniques described in the following subsections.

Before we begin our discussion on the techniques used for performance analysis, we need to identify the main factors that make up the execution time of a program. There are two main components of the execution time of a program [7]:

1. Execution path of the program
2. Execution time of each of the instructions in the execution path.

Following are two of the widely used techniques of performance analysis.

2.9.1 Program Path Analysis

A program could have numerous feasible execution paths of different execution times. In determining estimated bound of a program, these execution paths should be studied and those that result in extreme cases (best and worst case) execution time are to be considered. To do program path analysis, the following conditions are necessary in the program [7]:

- There should be a bounded number of iterations in loops
- There should be no recursive function calls.
- There should be no dynamic function calls.

As stated above, the main problem in program path analysis is that there are too many possible execution paths. This can be overcome by reducing the number of paths wherever possible. For e.g., in a conditional statement (*if-then-else*) the execution time for the true statement can be compared with that of the false statement and for a worst case analysis, the longer one can be considered. However, this may result in an overly pessimistic estimate. Moreover, pieces of code that are distinct in the program might still be connected logically.

Consider the following e.g.:

```
If (condition)
 Var  <------- 1 -- S1
Else
 Var  <------- 0 -- S2
Endif

If (Var)
 S3
Else
 S4
Endif
```

If we consider the false statement to take more execution time, then in the above example, the total estimated time would be `S2 + S4`. However, logically, `S2` and `S4` can never both be executed. Hence, it becomes necessary to do a data flow analysis of the program.

In doing program path analysis, the constraints contributing to timing include: *structural constraints*, which are concerned with the flow of control, and *functionality constraints* that are related to the program's logical flow.

Determining the maximum cost function for a program is a linear programming problem that can be solved by an ILP solver when it is fed with the combined set of functionality constraints, in conjunction with the structural constraints. The ILP solver has to maximize the cost function based on these constraints.

2.9.2 Microarchitecture Modeling

Microarchitecture modeling involves the modeling of extreme case execution times of sequences of instructions. These values are passed to the program path analysis to

determine the estimated bound of the program. This calls for a very detailed knowledge of the microarchitecture of the processor, memory system and peripherals. There are a number of new and innovative systems these days that improve the performance of pipelined execution units and caches. These tend to have an improved average case performance. However, they introduce the problem of variance of instruction execution times and make extreme case timing analysis difficult. To overcome this difficulty, new techniques are required to model pipelines and caches. [7] presents such new approaches.

The difficulty with estimating the execution time when cache is present is that at any point, a cache hit or a cache miss may result, and the course of execution is dependent on this; hence, the execution time of the program also depends on which of the two happens. One way of modeling cache then, would be to consider the cache hits and cache misses, and take a summation of them. The number of cache hits is multiplied by the time taken by an instruction that's encountered a cache hit takes to execute. A similar product is found for instructions that result in cache misses and these are added up, and the summation of all such instructions gives an estimated time. For this, it would make sense to group instructions logically in such a way that instructions that will result in cache hits are all grouped together, and those that will encounter cache misses are grouped together. But this is not always applicable to instructions in a single logical block within the program, because once a cache miss happens, a block is brought into cache and so if the next instruction is to be accessed next, it will not cause a cache miss. Thus, for every block only the first instruction will result in a cache miss. So we can partition every logical block of code into two sets – one corresponding to miss, and another, to hit.

The difficulty with estimating the execution time when pipelines are present is that the instructions in the pipeline may interfere with each other. They may be overlapped, but if one instruction is dependent on the completion of execution of a preceding instruction, then it has to wait until the previous instruction finishes execution. This is called *data hazard*. While this instruction is waiting, none of the subsequent instructions happen, and this is called as a *pipeline stall*. In the case of a branch instruction, if the branch happens to be taken, then at least one instruction will need to be flushed out of the instruction pipeline and this is known as *control hazard*. To model pipelines, we start off by assuming that the pipeline is empty, and so no pipeline stall occurs. We then add up the execution times of each of the instructions in the block, and then after simulating the execution of this block, we add up any possible delays due to pipeline stalls, if further instructions are brought into the pipeline.

2.9.3 Performance Evaluation in our system

Since the goal of this research is assessment of the relative overhead introduced by object-oriented programming, we need to compare the performance of our kernel with that of the non-OO version of the kernel. The method adopted here is to use the profiler provided by the simulator used to develop these kernels. This gives a fairly good idea of the overhead in terms of the number of instruction cycles and an analysis of the results

can help us arrive at the overhead in each category of operations, such as that in scheduling, that in computation etc..

Although there has been extensive work in the area of object-oriented operating systems as well as embedded real-time operating systems, the design and development of an object-oriented embedded kernel specifically for dataflow applications running on PEBB systems is a new and unique contribution that our research has made to the world of real-time embedded software. The research serves to combine the advantages of the OO paradigm and the dataflow computational model in achieving a high-speed kernel for an embedded real-time system.

Chapter 3

Kernel Architecture

This chapter describes the design of DARK++. It provides insight into the features an application designer needs to understand in order to write ECOs and use DARK++ for an application. It also discusses the rationale behind the most critical design decisions. This chapter presents the most important classes in the DARK++ system, including their responsibilities, interactions and the interfaces they provide. The following section is a discussion of the features of dataflow applications and the requirements that the dataflow model imposes on the kernel. Section 3.2 provides a brief overview of some specific kernel features. Section 3.3 describes the DARK++ classes, their role in the system, and how the various components interact with each other. Section 3.4 details the various kernel features. We first explain thread management in DARK++. This subsection explains the various states of the process (ECO) threads and includes a thread state diagram that indicates the basic thread management framework. This is followed by a discussion of the way context switching has been implemented in DARK++, which makes it faster than most other RTOSes. Subsequent subsections describe DARK++ time management, interrupt handling, mutual exclusion and real-time support respectively.

3.1 Dataflow Requirements

Section 1.3 introduced the performance issues associated with dataflow architecture.

Dataflow architecture is characterized by [32]:

1. *A large number of components*
   An embedded software system can be divided into components, which in turn can be encapsulated into processes. However, in the dataflow model, and specifically, in a PEBB system, every node (component) in the system is meant to be standardized, modular and reusable. Therefore such a system inherently means more number of components, and correspondingly, more number of processes in the software system.

2. *High frequency of inter-component communication*
   The many components in a dataflow system need some way of communicating with each other. This introduces the requirement of an elaborate and efficient inter-component communication, which is brought about by the use of data channels, as discussed in Section 1.1.

3. *Special scheduling requirements*
   Due to the data-driven nature of dataflow applications, processes need to be scheduled not only based on their relative priorities, but also the status of their incoming data channels. Moreover, the goal is not to start a process after every

incoming data token. Rather, processes are to be executed only after all their required incoming data channels have data in them.

The main requirements imposed on the underlying kernel by dataflow applications are as follows:

1. *High performance components*
   Since the hardware components in PEBB systems are replaced by their equivalent software components, and these form part of a real-time system, these software components should be minimal overhead components.

2. *Efficient Context Switching*
   As already mentioned, dataflow applications have more number of components than equivalent systems that follow other traditional software architectures. This means more number of context switches among the processes. Hence the context switching in these systems has to be very efficient.

3. *Efficient Inter-Component Communication*
   Due to the large number of components and the data-driven nature of dataflow applications, a very large fraction of the execution time comprises inter-component communications. Therefore it has to be highly efficient.

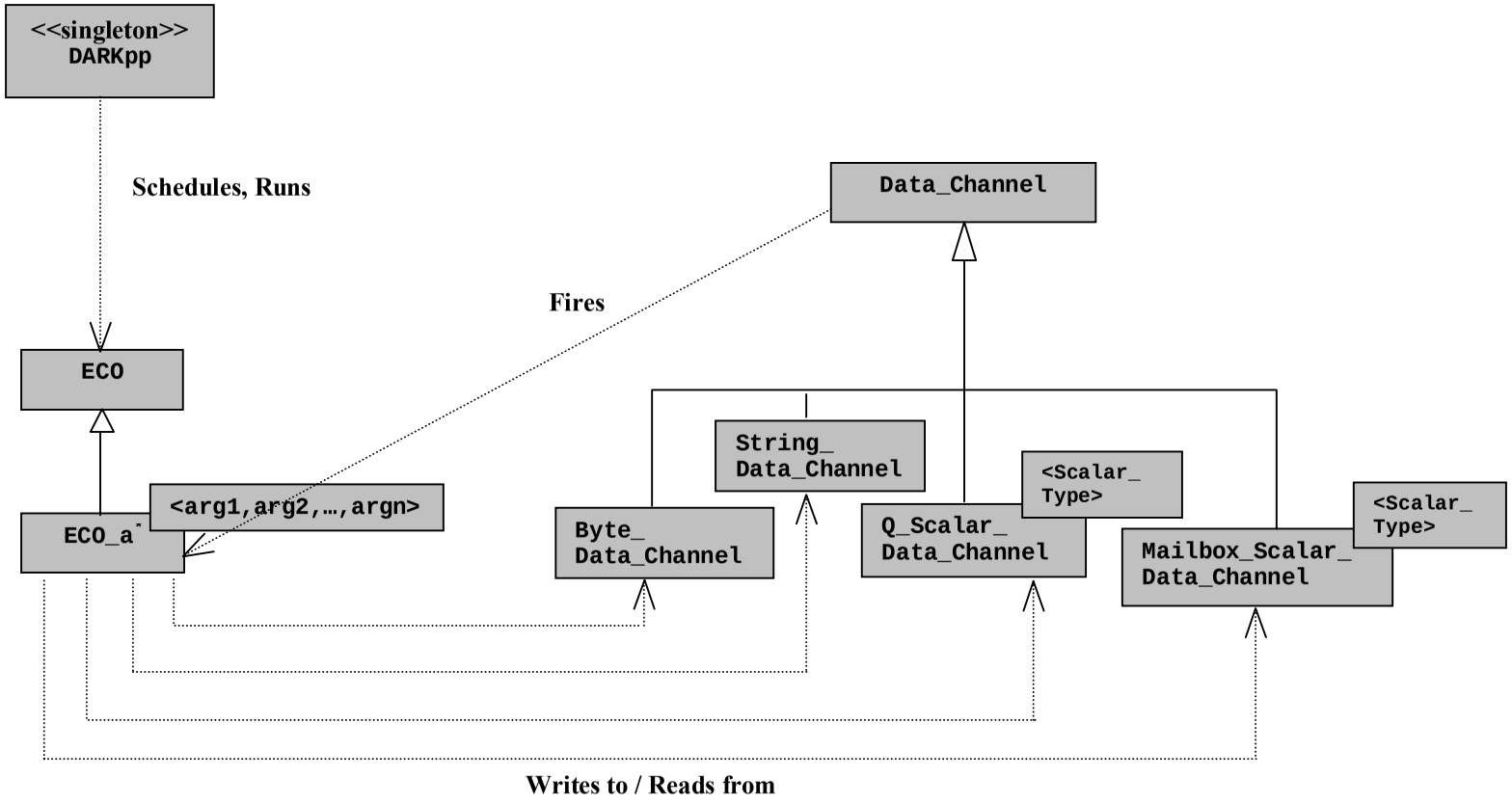
3.2 DARK++

DARK++ is a high-performance object-oriented kernel that addresses the requirements imposed by dataflow. It aims to reduce the overhead due to the use of OO programming (OO performance issues have been discussed in Chapter 1).

DARK++ is a preemptive, multi-threaded kernel. It always runs the highest priority thread that is ready. An executing process is preempted if a higher priority process is found to be ready when the currently running process makes a call to a kernel API.

DARK++ implements efficient context switching by taking advantage of *dual-register-set* hardware and saving and restoring only the required registers as opposed to all of them whenever this is feasible. This is discussed in Section 3.4.2.1. DARK++ also provides support for dynamic priorities, firing rules for specifying the data channel conditions necessary for process wakeup, and typed data channels for efficient and reliable inter-process communication.

Figure 3.1. DARK++ System Diagram
* - Sample ECO called ECO_a – the specific ECO name goes here



DARK++ is implemented in C++, with a few key elements- context switching, dual register set support, and interrupt handling written in assembly. Because it is intended for embedded power electronics control, it currently runs on Analog Devices SHARC 21xxx 32-bit digital signal processors. Dataflow processes, or ECOs, are implemented as C++ classes. DARK++ uses a statically initialized set of ECO objects, together with a statically initialized set of interconnecting data channel objects.

3.2.1 DARK++ Classes

The major classes that comprise the DARK++ system are: `DARKpp`, `ECO`, `Data_Channel` and some helper classes that form kernel's internal data structures. The `DARKpp` class is a singleton, which means there is only one DARKpp object (the kernel object) in the system. It is responsible for scheduling and running the processes, which are represented as `ECO` objects. The `ECO` class is a base class from which templates for specific ECOs are derived. These templates take the types of their input ports as parameters. Since there are two types of data channels – the queued and the mailbox, which are in turn implemented as templates derived from the base `Data_Channel` class, the input port type parameters for specific ECOs help specify which of the two types of data channel each port of an ECO is. This can be tailored to suit the specific application; i.e., the application designer can instantiate an ECO appropriately. The ECOs read data from their input data channels, perform necessary computation and write data to their output data channels. The data channels, on receiving data, fire the next process, by adding their sink ECO to the ready queue, which is an internal data structure that the system uses.

Figure 3.1 gives the "big picture" of the system. It shows the most important relationships among the main classes and the interaction amongst the respective objects.

3.2.1.1 Class DARKpp

The `DARKpp` class is the kernel class. It performs scheduling of the processes and makes use of the *ready queue* and *event queue* data structures, which are also implemented as classes (`Ready_Queue`, `Event_Queue` respectively).

The kernel, in its every iteration, checks for pending interrupts and handles them if there are any. It then executes the next ready process from the ready queue if this process has a priority greater than or equal to the currently running process. Thus, it preempts the current process if it finds a new process of greater or equal priority. Preemption on encountering an equal priority ensures fairness by preventing a process from hogging the processor for a long period.

The kernel uses the variable `actions_pending` whenever it needs to check whether any timed events have to be performed/pending interrupts have to be serviced. This variable can have one of three values:

- `no_actions`, which indicates that there are no pending timed events/interrupts
- `future_actions`, which indicates pending timed events
- `current_actions`, which indicates pending interrupts

Since there has to be a single kernel object in the system, a static method that returns a static reference to a kernel object is used. This method, `getInstance()`, is shown in Figure 3.2.

```
DARKpp& DARKpp :: getInstance()
{
   static DARKpp the_kernel;
   return the_kernel;
}
```

Figure. 3.2 The getInstance() method

3.2.1.2 Class ECO

The `ECO` class is an abstract base class for ECOs containing the ECO implementation method as a virtual method, which can be defined in the particular ECO subclass, since ECOs are functionally distinct. The ECO objects in the system are the processes that are scheduled and executed by the `DARKpp` kernel.

An ECO designer should take the following steps to write a class for a specific ECO:

- Call the base `ECO` class constructor from the constructor of the new class with the following parameters: number of input and output ports, the firing rule for the ECO, the initial priority, an array of pointers to the input ports and an array of pointers to the output ports.
- Set the specific configuration information for the ECO in the new class's constructor.
- Write the implementation function and any other new functions if required.

Figure 3.3 shows the ECO class interface. It includes comments that explain the role of every data member/method.

```
class ECO
public:
Implementation() = 0; // pure virtual function
Wait_To_Fire();       //returns true or returns to OS
Register_OS();    // registers with DARKpp
Current_Priority();   // returns current ECO priority
SetPriority(Priority ); // sets ECO priority
SetIn_Ports_Ready(Firing_Mask );//set mask after write
Timed_Wait_To_Fire(int );
Delay(int );
In_Ports_Ready();  // returns mask showing ready ports
SetProcess_State(ProcessState );
Blocked();    // returns true if process is blocked
SetWakeup_Call(Firing_Mask );  // sets the mask that fires the ECO
Wakeup_Call();     // returns the mask that woke it up
Process_State();      // returns process state
FIRE_Rule();     // returns firing rule
Get_Heap_Position();    // gets process's position in ready queue heap
Set_Heap_Position(int );// used to alter position of ECO in ready queue
swap(ProcessState );  // changes process state and goes to OS
ECO_Env();    // returns context information for the process
Stack();  // returns pointer to the stack in which process is running
StackSize();     // returns size of process stack
protected:
Register_As_Source(int ); // register as source ECO of o/p port(s)
Register_As_Sink(int );   // register as sink ECO of i/p ports(s)
```

Figure 3.3 ECO Class Interface

3.2.1.3 Class Data_Channel

The `Data_Channel` class is a base class for all types of data channels. From this class, we have a derived `Q_Scalar_Data_Channel` and `Mailbox_Scalar_Data_Channel` template classes. While `Q_Scalar_Data_Channel` represents queued data channels, which can contain multiple data items (specified in the constructor), while `Mailbox_Scalar_Data_Channel` represents mailbox data channels, which can contain a single data item. It is more efficient to implement mailbox data channels separately so that we can do away with queue arithmetic.

Besides these, the `String_Data_Channel` and `Byte_Data_Channel` classes are provided. If required, specific data channels for user-defined data types may be defined as subclasses of the `Data_Channel` class by using *traits*. The need for a `Data_Channel` base class and template subclasses for scalar data channels, rather than a single template base class arises because ECO objects need to store pointers to their input and output data channels (so as to *read*/*write* from the appropriate data channel). It would not be possible for an ECO object to contain an array of `Data_Channel` pointers if `Data_Channel` were defined as a template base class. This necessitates having a `Data_Channel` base class that represents any type of data channel.

As mentioned, the ECO objects store an array of *pointers* to `Data_Channel` objects rather than an array of `Data_Channel` objects. The rationale for this is that a data channel, essentially being an interconnection between two ECOs, forms the output data channel for one ECO and the input to another. Since both the *source* and the *sink* ECOs require a knowledge of the identity of this data channel, we would need this `Data_Channel` object in the *output data channel* array of the source ECO as well as in the *input data channel* array of the sink ECO, which is not possible. Therefore references to the `Data_Channel` objects (`Data_Channel` object pointers) are stored instead.

Data channels may be interrupt-driven or non-interrupt-driven. While interrupt-driven data channels are input data channels to processes that are fired by interrupts, the non-interrupt-driven data channels are written to by their source processes. This information (interrupt-driven or not) is specified in the constructor to the data channels.

Figures 3.4 through 3.7 show the interfaces provided by all the data channel classes.

```
class Data_Channel

public:
Capacity();    // max. no. of entries
Entries();     // current no. of entries
Available_Capacity();
Flush(int ); // remove given no. of entries from end of queue
Blocked();  // returns true if blocked

protected:
Register_OS();   //registers with DARKpp
setBlocked();
resetBlocked();

/* Read_bytes pass char pointer to read, no. of bytes to be read */
Read_bytes(char* , int );

/* Write bytes pass char array to be written, no. of bytes */
Write_bytes(char* , int );
```

Figure 3.4 Data_Channel Base Class Interface

```
template Q_Scalar_Data_Channel<Scalar_T> :: public Data_Channel,
template Mailbox_Scalar_Data_Channel<Scalar_T> :: public Data_Channel
public:
Read(Scalar_T& ); // pass reference parameter of appropriate
                  // type to read
Write(Scalar_T ); // pass parameter of appropriate type to
                  // write
```

Figure 3.5 Scalar_Data_Channel Template Class Interface

```
class String_Data_Channel :: public Data_Channel

public:
      Read(char* );  //pass char pointer to read in string
      Write(char* ); //pass char array write
```

Figure 3.6 String_Data_Channel Class Interface

```
class Byte_Data_Channel :: public Data_Channel

public:
/* Read_bytes follows: pass char pointer to read, no. of bytes
   to be read */
Read_bytes(char* , int );

/* Write_bytes follows: pass char array to be written, no. of
   bytes */
Write_bytes(char* , int );
```

Figure 3.7 Byte_Data_Channel Class Interface

3.3 Client Code

Having discussed the important classes and the framework of the system, we now explain how these can be used by an application designer to run a dataflow control application.

We consider the closed-loop three-phase inverter as an example control application and show how it can be run using our kernel. Following is the dataflow graph for the application.

This application comprises 9 ECOs and 20 data channels. To run this application, the data channel objects first have to be declared. Following this, the ECO objects are declared with the association amongst them being established by providing the input and output port information in the constructors of the ECOs. The constructor takes the other necessary information as well, such as the configuration information for the ECO, the firing rule, the initial priority of the ECO and the size (in bytes) of stack space required to run the ECO.

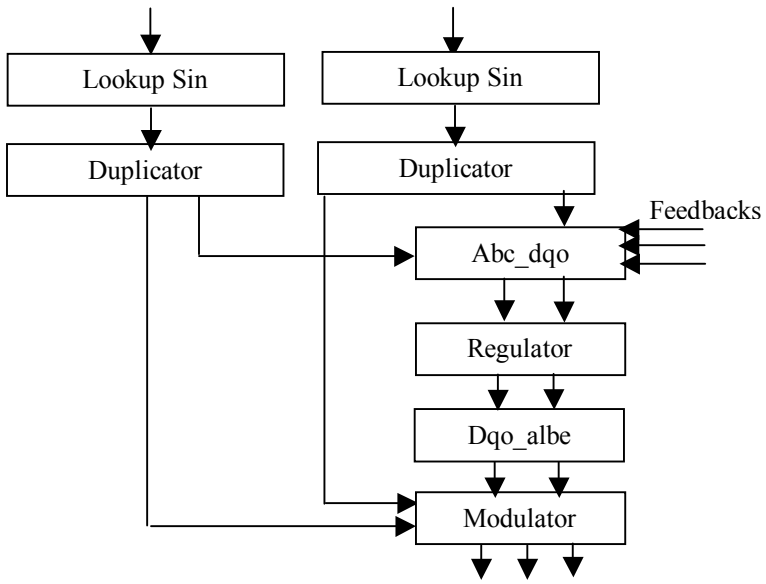


Figure 3.8. Closed-loop three-phase inverter

```
template <class Bool_Data_Channel_i, class Float_Data_Channel_o >
void Adc_Va <class Bool_Data_Channel_i, class Float_Data_Channel_o>
                          :: Implementation(){ do
  {  switch (wakeup_call)
     {
         case ADC_VA_FIRING_MASK_DEFAULT:
                                              default_action();
                  break;
        case ADC_VA_FIRING_MASK_EXCEPTION:
                                              exception_handling();
                  break;
     }
  }while (Wait To Fire());
```

Figure 3.9. Template for ECO Adc_Va

```
void default_action()
{  /* Input variable */
   bool start;

   /* Output variable */
   float va;

   int Adc_offset;
   float Adc_scale;
   float Va_offset;
   float Va_scale;

   /* Intermediate variable */
   int Adc_value;

   Adc_offset = confg.Adc_offset;
   Adc_scale = confg.Adc_scale;
   Va_offset = confg.Va_offset;
   Va_scale = confg.Va_scale;

   /* Read input from data channel */
   Adc_Va_Start->Read(start);

   Adc_value = *(int *)confg.Data_buffer;
   Adc_value -= Adc_offset;

   va = (float)(Adc_value) * Adc_scale;

   va = (va-Va_offset) * Va_scale;

   va = 60.0;

   /* Update output data channel */
   Adc_Va_Va->Write(va);
}
```

Figure 3.10. Default action for ECO Adc_Va

Figure 3.9 shows the template code for a sample ECO – the `Adc_Va`. As we can see, it takes the types of the input and output ports as parameters. The implementation body is a *while* structure that fires the ECO again if it has data in its input port. Based on which firing mask triggered the ECO (`wakeup_call`), the appropriate action is performed. These actions are also provided for the ECO. A sample function is shown in the figure below

3.4 DARK++ Kernel Features

3.4.1 Thread Management

An ECO can be viewed as a process that executes its `Implementation` code provided by the ECO designer. The various possible states of these processes are: *ready*, *run, blocked*, *wait_for_fire*, *timed_wait*, *timed_wait_for_fire* and *dead*. When the kernel starts, each thread is in the *wait_for_fire* state. A process is in *ready* state once its required input data channels have data tokens in them. The ready process of the highest priority is run by the kernel and such a process (an executing process) is in the run state. The process is *blocked* when it tries to write to a full data channel.

After every *read* operation on a data channel, the status of the source ECO (ECO that writes to this data channel) is checked. If the source ECO is found to be blocked, then it is unblocked. Similarly, after every *write* operation, the mask of its sink ECO (ECO that reads from this data channel) is updated; i.e., the bit corresponding to the data channel in question is set. Thus, while a *read* operation could unblock a process blocked on a data-channel, a *write* operation could fire it.

The `wait_to_fire` function can be used to fire the ECO again. If the ECO is not ready for firing, it goes into the *wait_for_fire* state. The user can also delay the execution of the ECO for a pre-determined time, which puts the ECO into *timed_wait* state. The *timed_wait_for_fire* state is a combination of *wait_for_fire* and *timed_wait*. An ECO in this state can be fired if a firing mask becomes true *or* if the time period elapses. The ECO goes into the *dead* state once it finishes execution.

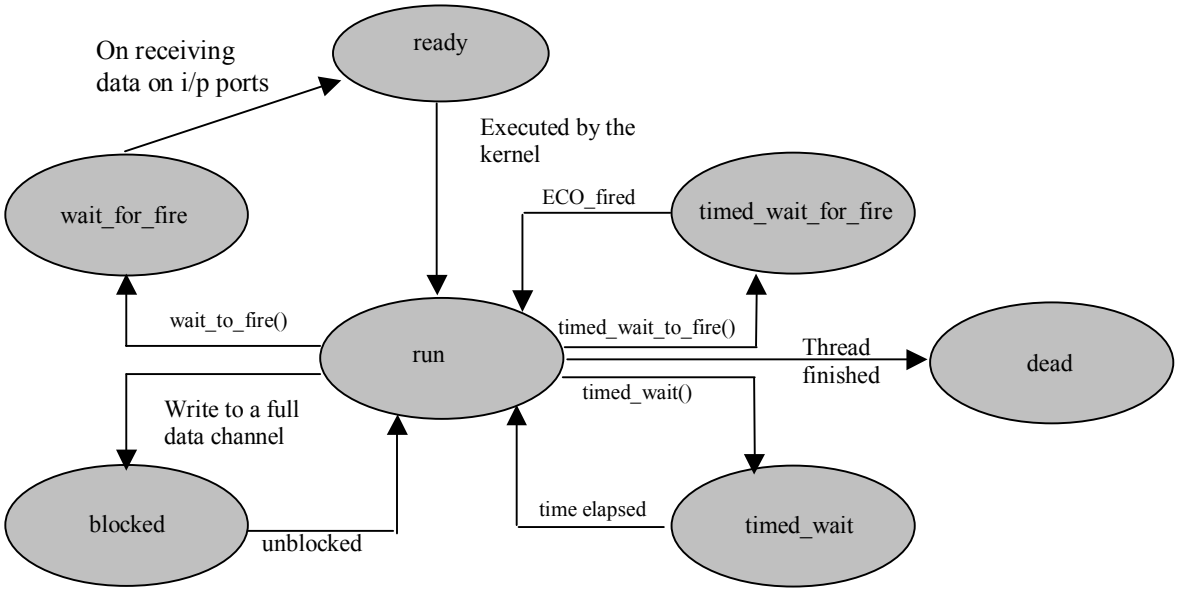


Figure 3.11. Thread state diagram

3.4.2 Context Switching

Operating systems can have one of two types of schedulers:

- Active Scheduler
- Passive Scheduler

An active scheduler runs as a separate thread and therefore necessitates a context save and restore (of the status of all the registers) every time there is a transfer of control between the scheduler and a process. A passive scheduler, on the other hand, does not run as a separate thread and is called by the process threads (through normal function calls). Although the passive scheduler approach obviates the need for explicit context save and restore, thus making it faster, this approach does not allow for preemption because if there has been a transfer of control from a process to the scheduler through a function call, the scheduler cannot suspend the currently running thread if need be, to run a new higher priority process. Control simply has to go back to the process thread from the scheduler. DARK++ therefore uses the active scheduler approach, in which context switches are brought about by the use of calls to the `setjmp` and `longjmp` functions. In the normal `setjmp` and `longjmp` calls, the context is entirely saved and restored, respectively. This means the contents of all the registers in the processor are saved during a `setjmp` and restored during a `longjmp`. However, DARK++ exploits the *dual-register-set* hardware provided by the Analog Devices SHARC 21160 micorprocessor for a substantially more efficient context switching. This approach is detailed in the following subsection.

3.4.2.1 Dual-Register-Set Hardware and High Speed Context Switching

Many digital signal processors used in embedded control systems, ADSP 21160 being one, have two sets of registers for increased performance – the *primary* set and the *alternate* set. DARK++ uses the primary register set for the kernel and the alternate register set for the process threads. Due to the use of two independent sets of registers for the kernel and the process threads, all that is required during a transfer of control between the two is flipping of a bit in a control register, which denotes the current *mode* (indicating whether the currently running thread uses primary set/secondary set), and saving/restoring some key status registers. DARK++ uses customized *setjmp* and *longjmp* assembly language procedures that selectively save/restore just these required registers.

Since most context switches in dataflow applications occur between the scheduler and executing threads, minimizing the cost of such switches increases the performance significantly. The use of the dual-register-set architecture in DARK++ for high-speed context switching between the scheduler and application threads has been found to reduce the switching time by 80% [32].

3.4.3 Time Management

DARK++ provides APIs to allow ECOs to request a timed delay. In most other RTOSes, the kernel checks each waiting thread at every clock tick, and adds it to the ready queue when the waiting period has expired. However, this technique can introduce unnecessary overhead if there are a number of waiting threads. Hence DARK++ uses a different approach to handle timed delays. When the `timed_wait()` or `timed_wait_for_fire()` method is called, the delay is converted into an absolute time by adding the current system time to it and then stored in the ECO (process) object. The thread is then added to the waiting queue in which the threads are arranged in ascending order by absolute time and `actions_pending` is set to `future_actions`.

The kernel checks for `actions_pending` and adds the process back to the ready queue when the deadline has expired. To check whether the deadline has been reached, it compares the system time with the thread wakeup time of the first thread in the waiting queue. The scheduler needs to check only the first thread in the waiting queue unless that thread's waiting period has elapsed.

3.4.4 Interrupt Handling

There are many RTOSes that support interrupt handling through the use of compiler-provided mechanisms, using C functions that can be used as interrupt routines. This method involves a substantial overhead in context switching, since all registers are saved and restored while handling interrupts. The C compiler provided by Analog Devices for its SHARC DSPs supports this approach, and in addition, also provides the option of using the alternate register set for interrupt handling (since the C runtime uses only the primary register set). However, DARK++ cannot use this option, since it uses both the alternate and primary register sets, as explained in Section 3.4.2.

DARK++ uses an alternative approach for handling external interrupts. This method provides performance comparable to that of using the alternate register set for interrupt handling. Here, rather than placing actions directly in the interrupt handler itself, DARK++ uses a minimal footprint handler that simply logs incoming events into the *event queue*, which is managed by the DARK++ scheduler. The interrupt handler runs in the currently active register set and only needs to save and restore a couple of registers. It logs a 32-bit code representing the interrupt that was received, into the event queue (a circular buffer of incoming events) and then returns control to the kernel. The status of the event queue is reflected by the `actions_pending` variable that we have already explained.

DARK++ also supports clock interrupts and non-maskable interrupts (NMI). The clock interrupt ISR is written in assembly and simply increments the kernel data member, `current_time` that is used for time management. Only a few registers required for incrementing a variable are saved and restored in this ISR. NMI is used for emergency condition notification and requires a time critical response. In most cases, it results in a call to the application's emergency shutdown procedure, bypassing all other kernel as well as application code.

3.4.5 Mutual Exclusion

Since threads have no shared memory and communicate only through data channels, most mutual exclusion problems do not arise in DARK++. However, there is one condition that needs to be handled. If interrupts occur when a process is executing, then control goes to the kernel. Under such a circumstance, it is important to ensure that the kernel resumes and completes the execution of the process thread that was suspended due to the interrupt, as soon as it executes the interrupt handler. This is because the event associated with the interrupt might have caused a higher priority process to be triggered. If this higher priority process is allowed to preempt the suspended process, then there is a possibility for data to be corrupted if the two processes access a common data channel. In such a scenario, however, the interrupt handler writes a 32-bit code into the event queue and control returns to the interrupted thread so that the event associated with the interrupt actually gets executed only after control returns to the kernel thread.

3.4.5.1 Volatile Declarations

In the embedded system context, it is imperative to have an efficient and optimized executable. With the optimizer enabled, typically several variables are cached in registers to make data fetches more efficient.

As explained in Section 3.4.2.1, DARK++ uses the dual-register-set hardware for efficient context switching. This means that the kernel and the process threads work with distinct registers. Consequently, the two threads use different registers to cache the same data and this could lead to inconsistencies. It should be noted here that if all methods that manipulate data members private to their class were called through normal method call, then this problem does not arise, as these data members would not be cached in their callers. However, in the interest of our high performance objective, methods are all inlined, and this causes even the private data to be cached in registers, as part of the caller's thread. Therefore it is critical to identify all data that can potentially be accessed by both the kernel and the processes threads, and declare them *volatile* to ensure that they always get accessed from the memory instead of from registers.

The `front` and `rear` data members of the `Data_Channel` class, and the `in_ports_ready`, `wakeup_call` and `process_state` data members of the ECO class are declared volatile since these are accessed in the *Read*/*Write* operations, which can potentially be called from the kernel thread and the process thread.

In the context of interrupt service routines (ISRs), an important fact to consider is that interrupts could occur at anytime during the execution of the kernel/process thread and so if the ISR shares any data with either of these threads, then such data has to be declared

volatile in order to avoid the executing thread from using an incorrect value that was cached in a register prior to the occurrence of the interrupt, after control returns back from the ISR. The `actions_pending` variable is declared volatile for this reason.

3.4.6 DARK++ Configurable Options

The DARK++ kernel can be configured to yield four distinct versions. These are obtained by selectively retaining/removing certain kernel features. Removal of features leads to an increase in performance with a concomitant reduction in run-time flexibility. The application designer can select the most appropriate DARK++ version for a given application's requirements. These are compile-time configurable by preprocessor macros. Table 1 lists the features in different versions.

The *full-featured* version of DARK++ has nothing disabled, and is a multi-threaded preemptive kernel. This version of the kernel schedules threads dynamically based on their firing rules and priorities. After every OS call (*read* and *write* operations), the scheduler is invoked to check for higher- and equal-priority threads. A context- switch takes place if and only if a higher- or equal- priority thread is ready. Preemption by an

equal-priority thread ensures fair scheduling. This version is enabled by the preprocessor directive `PREEMPTIVE_MTHREADED`. It should be noted here that DARK++ offers preemption in a restricted sense, rather than the more common "textbook" sense, wherein preemption means stopping an executing thread as soon as a higher priority thread becomes ready. Here preemption happens only during API calls. However, the dataflow paradigm rules out the possibility of a higher priority process becoming ready while the current process is executing. This is because a process can become ready only when its source process has written data to its input data channel, but the source process thread could not have been running in conjunction with the currently executing thread.

The *non-preemptive* version of DARK++ does not invoke the scheduler on every OS call but instead, runs each thread until the thread suspends itself waiting for input data. Thus we avoid the overhead of calling the scheduler after every *read* and *write* operation by sacrificing immediate response to higher-priority threads. This version is enabled by the preprocessor directive `NONPREEMPTIVE_MTHREADED`.

The other two versions of DARK++ are single-threaded and avoid the time spent in context switching, thereby giving a significant performance boost to the application. The single-threaded approach is ideal for monotonic applications that execute sequentially, but unsuitable for applications that are highly dynamic. While the dynamically scheduled single-threaded version of DARK++ uses firing rules and priorities to select a process for execution, the statically scheduled single-threaded version uses a pre-computed firing order for threads, eliminating all use of priorities and firing rules, and is therefore the fastest. The single-threaded dynamically scheduled version is enabled by the preprocessor directive `SINGLETHREAD_DYNSCHD` and the single-threaded statically scheduled version is selected by `SINGLETHREAD_STATSCHD`.

These directives in turn use three other directives to control the features of the kernel – enable/disable preemption, enable/disable multithreading, and enable/disable dynamic scheduling. These are `PREEMPTIVE`, `MTHREADED` and `DYNSCHD` respectively.

As mentioned, data Channels are of two types – message queues and mailboxes. A mailbox is an inter-processor data channel and is nothing but a special case of message queue where the size of the queue is one. This obviates the need for queue arithmetic and is therefore more efficient.

Multithreaded/ Single-threaded	Kernel Version	Preemptive/ Non-preemptive	Dynamically Scheduled/ Statically Scheduled
`MTHREADED=1`	`PREEMPTIVE_MTHREADED`	`PREEMPTIVE=1`	`DYNSCHD=1`
	`NONPREEMPTIVE_MTHREADED`	`PREEMPTIVE=0`	`DYNSCHD=1`
`MTHREADED=0`	`SINGLETHREAD_DYNSCHD`	`PREEMPTIVE=0`	`DYNSCHD=1`
	`SINGLETHREAD_STATSCHD`	`PREEMPTIVE=0`	`DYNSCHD=0`

Table 3.1. Configurable options in DARK++

3.4.7 Real-time Support

Real-time scheduling algorithms could be based on fixed priorities or dynamic. While the rate monotonic priority assignment (RMA) algorithm, which assigns higher priority to shorter tasks is the optimal fixed-priority algorithm, the deadline driven scheduling algorithm is the optimal dynamic scheduling algorithm. The dynamic scheduling algorithms, however, in general, have a lot of overhead associated with them.

DARK++ provides the user the option of enabling real-time support. It has provision for the user to assign a function handle that will be used to run the provided scheduling algorithm. If this handle is null then the default algorithm that DARK++ uses is the fixed-priority RMA algorithm. In order to meet the high-performance objective, the complex real-time support necessary for POSIX compliance has been avoided in DARK++.

DARK++ provides the following simple API to monitor real-time deadlines.

A deadline for an ECO can be set using the method:

```
void ECO :: set_deadline(int time);
```

The `time` parameter specifies the time by which the ECO has to finish its execution. When this method is invoked, `time` is converted to an absolute time by adding the current system time to it and the ECO is added to the *deadline queue*.

The following method can be used to ascertain whether an ECO has met its deadline:

```
bool ECO :: check_deadline();
```

This method removes the ECO from the deadline queue and returns true if the ECO met its deadline.

The DARK++ scheduler checks the first entry of the deadline queue in each switching cycle. If it finds an ECO that missed its deadline, it calls a user-provided handler.

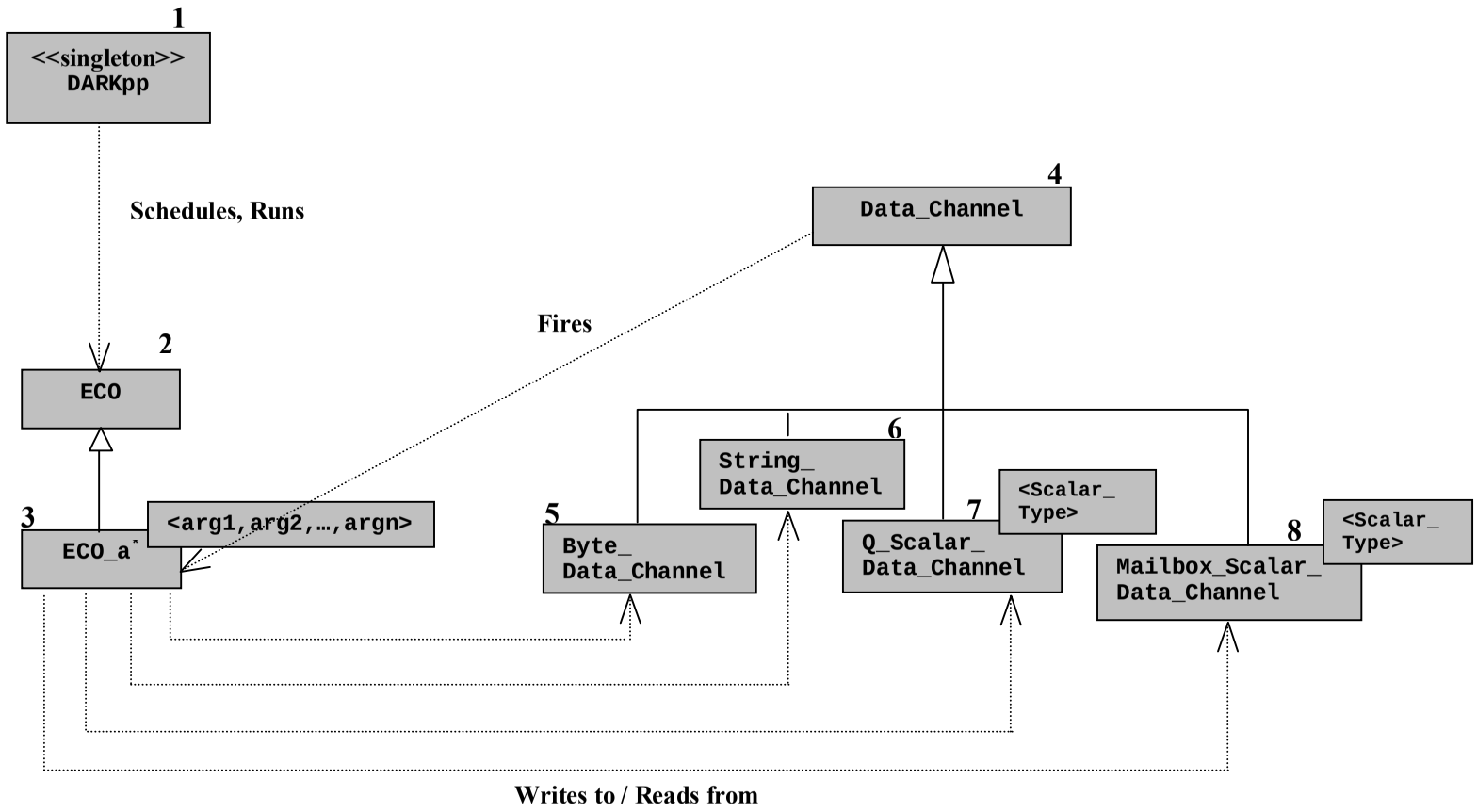
Chapter 4

Kernel API

This chapter discusses the various application programming interfaces provided in the DARK++ system. The primary operations may be classified as those for inter-ECO communication, data channel management, thread management and time management. All these operations are provided as methods in their respective classes.

We revisit the system class diagram (Figure 4.1). In this diagram, classes are numbered (1 – 8) and the interfaces provided are explained class-wise. The methods of any class labeled x are discussed under section 4.x.

Figure 4.1. DARK++ System Diagram
* - Sample ECO called ECO_a – the specific ECO name goes here



4.1 Class DARKpp

```
DARKpp(int num_processes, int num_data_channels);
```

Parameters

num_processes: The total number of ECOs in the application run using DARKpp kernel.

num_data_channels: The total number of data channels in the application run using DARKpp kernel.

Description

This is the constructor for the DARKpp kernel object. It is used to create the single kernel object. It allocates memory for storing the specified number of ECO and Data_Channel objects in its internal OS_All_processes and OS_All_Data_Channels array data members and also stores the number of ECOs and number of data channels in the kernel's internal data members num_ECOs and num_DCs respectively.

```
static DARKpp& getInstance(int num_processes,
                           int num_data_channels);
```

Prototypes

num_processes: The total number of ECOs in the application run using DARKpp kernel.

num_data_channels: The total number of data channels in the application run using DARKpp kernel.

Description

This method is used for obtaining the reference to the singleton kernel object. It calls the DARKpp constructor the first time it is called and returns a reference to the created (*static*) kernel object every time it is called.

```
void  execute();
```

Description

This is the main part of the kernel. It does the scheduling. It checks to see which of the versions of the kernel has been chosen (in DARKpp_cfg.h), namely, full-featured, preemptive, single-threaded statically scheduled and single-threaded dynamically scheduled, and does scheduling appropriately.

4.2 Class ECO

```
ECO(Inports_Num in, Outports_Num on, FiringRule FR,
      Prty IP, Stack_Size stk_size, Data_Channel** iports,
      Data_Channel** oports);
```

Parameters

`in`: integer specifying number of input ports to the ECO.
`on`: integer specifying number of output ports from the ECO.
`FR`: firing rule (array of firing masks, each of type `Priority_Firing_Mask`) of the ECO.
`IP`: integer specifying initial priority of the ECO.
`stck_size`: integer specifying required stack size for the ECO in bytes.
`iports`: Array of `Data_Channel` pointers that are inputs to the ECO.
`oports`: Array of `Data_Channel` pointers that are outputs from the ECO.

Description

This is the constructor for the `ECO` base class and is called from the specific subclass constructors. It creates the `ECO` object and initializes it by storing the number of input ports and number of output ports in its respective internal data members, assigning firing rule, initial priority and stack size to respective data members. It also allocates memory for the stack in which the `ECO` process is to run with the specified `stck_size` number of characters. Following this, it registers the `ECO` object as the sink ECO of all its input data channels and source ECO of all output data channels and also registers with the `DARKpp` kernel object.

```
bool  Wait_To_Fire();
```

Description

- Used to fire an ECO.
- The ECO calls this method to fire itself again.
- Returns true if there is data in the incoming data channels. Else, it suspends the ECO by setting its state to *wait_for_fire* and returns control to the scheduler thread.

```
Prty Current_Priority();
```

Description

This method returns the current priority of the ECO process.

```
void SetPriority(Prty prty);
```

Parameter
`prty`: integer parameter specifying required priority of the ECO.

Description
This method sets the priority of the ECO process to the specified value. If an ECO calls this method on itself, control returns to the ECO after the priority is changed.

```
bool Blocked();
```

Description
This method returns true if the ECO is blocked waiting for data and false otherwise.

```
bool  Timed_Wait_To_Fire();
```

Description
This method causes the ECO to wait for a firing rule to go true *or* delays for a certain time. Two conditions can remove this block:

- A firing rule goes off.
- The timed wait expires.

```
bool delay(int delay_time);
```

Argument
`delay_time`: integer parameter specifying required delay.

Description
This method delays the ECO process for the specified time. It is similar to the sleep() function provided in many commercial RTOSes.

```
void SetDeadline(int time);
```

Argument
`time`: integer parameter specifying deadline for the ECO process.

Description
This method converts `time` to an absolute time and adds the reference to the `ECO` object to the deadline queue.

```
bool Check_Deadline();
```

Description
This method removes the ECO entry from deadline queue and returns true if the ECO process has met its deadline. Otherwise it returns false.

4.3 ECO subclasses

```
<eco_name>( FiringRule firing_rule, Prty priority
            ,Stack_Size  stack_size,
  [<eco_config_name>  eco_config_info],
  [<input_data_channeltype_1>*  input_data_channel_1,…..
    <input_data_channeltype_m>*  input_data_channel_m],
  [<output_data_channeltype_1>* output_data_channel_1,…..
    <output_data_channeltype_n>*  output_data_channel_n] );
```

- eco_name refers to the specific type of ECO, or the name of the required ECO subclass.
- eco_config_name refers to the type-defined structure for the ECO's configuration information. This argument exists for ECOs that need to store some specific configuration information and does not exist for others.
- For arguments input_data_channel_1…input_data_channel_m, m is the number of input data channels to the ECO, which varies from one ECO to another but is the same for a particular type of ECO.
- input_data_channeltype_1…input_data_channeltype_m correspond to the template parameter names for this ECO subclass template.
- For arguments output_data_channel_1…output_data_channel_n, n is the number of output data channels from the ECO, which varies from one ECO to another but is the same for a particular type of ECO.
- output_data_channeltype_1…output_data_channeltype_n correspond to the template parameter names for this ECO subclass template.

Parameters

firing_rule: firing rule (array of firing masks, each of type Priority_Firing_Mask) of the ECO.

eco_config_info: configuration information for the specific ECO. It is of type eco_config_name, which is a user-defined type representing the structure of the configuration information for the ECO.

priority: integer specifying initial priority of the ECO.

stck_size: integer specifying required stack size for the ECO in bytes.

The remaining arguments are `Data_Channel` pointers. Pointers to input data channels are specified first, followed by pointers to output data channels.

Description
This is the constructor for a specific type of ECO. It creates the required type ECO object and assigns input/output data channel pointers. It also stores configuration information in the internal data member. It also allocates memory for input and output port arrays and initializes the arrays to contain the passed data channel pointers – this gives every ECO knowledge about its input and output ports.
This constructor calls the `ECO` base class constructor with parameters `firing_rule`, `priority`, `stack_size` and pointers to the input and output data channel pointer arrays just created.

```
void Implementation();
```

This is defined as a pure virtual method in the base class `ECO`.

Description
The implementation method represents the code for the required computation/other function to be performed by the ECO. It typically comprises a loop that repeats as long as `Wait_To_Fire()` returns true, and a case structure within the loop that takes different actions based on the firing rule that woke the ECO process up.

4.4 Class `Data_Channel`

```
Data_Channel(int sz, int es, enum Overflow_Style os,
                     bool interrupt);
```

Parameters

`sz`: maximum number of bytes that can be stored in the data channel.

`es`: element size, or the size of each element that will be stored in the data channel. This depends on sizeof(<element_type>), where 'element_type' is the type of data that the data channel will be used to store.

`os`: one of: *blocked*, *overwrite_oldest* and *overwrite_newest*.

`interrupt`: true if the data channel is interrupt-driven and false otherwise.

Description
This is the constructor for the base class `Data_Channel` and is called by the subclass constructors. It creates data channel object. It allocates memory for `sz` number of

elements in data channel buffer and initializes the queue (buffer pointers) `front` and `rear` to 0. The constructor also registers the data channel with the DARKpp kernel.

```
int Capacity();
```

Description
This method returns the maximum number of data elements that can be stored in the data channel.

```
int Entries();
```

Description
This returns the number of data elements present in the data channel.

```
int Available_Capacity();
```

Description
This method returns the remaining number of data elements that can be stored in the data channel.

```
void  Flush(int num);
```

Parameter
`num`: required number of data elements

Description
This method removes the specified number of data elements from the end of the data channel queue (buffer).

4.5 Subclass `Byte_Data_Channel`

```
Byte_Data_Channel(enum Overflow_Style os, Cpcity cap,
                  bool interrupt);
```

Parameters
`os`: one of: *blocked*, *overwrite_oldest* and *overwrite_newest*.
`cap`: maximum required number of bytes in data channel.
`interrupt`: true if the data channel is interrupt-driven and false otherwise.

Description
This is the constructor for the `Byte_Data_Channel` class. It creates a byte data channel object by calling the base class constructor with element size 1.

```
void Write_bytes(char* data, int length);
```

Parameters
`data`: character (byte) block to be written.
`length`: size of the block to be written.

Description
This method writes the passed block of data into the data channel buffer by calling `Write_bytes` method in the `Data_Channel` base class. If the data channel is full, it checks the overflow style and overwrites data if it is *overflow_newest* or *overflow_oldest*; it blocks otherwise.

```
void Read_bytes(char* data, int length);
```

Parameters
`data`: character (byte) array to be read into.
`length`: size of the block to be read.

Description
This method reads the specified number of bytes from the data channel buffer into `data` array by calling the `Read_bytes` method in `Data_Channel` base class. The source process blocks waiting for data if this data channel is empty.

4.6 Subclass `String_Data_Channel`

```
String_Data_Channel(enum Overflow_Style os, Cpcity cap,
                    bool interrupt);
```

Parameters
`os`: one of: *blocked*, *overwrite_oldest* and *overwrite_newest*.
`cap`: maximum number of characters in data channel.
`interrupt`: true if the data channel is interrupt-driven and false otherwise.

Description
This is the constructor for the `String_Data_Channel` class. It creates a `String_Data_Channel` object by calling the `Data_Channel` base class constructor with the size parameter equal to `cap` and element size 1. So a total of `cap`

characters will be allocated. Element size will not be used since strings are of variable length.

```
void Write (char* data);
```

Parameter
`data`: character array(string) to be written.

Description
This method writes the passed string into the data channel buffer. If the data channel is full, it checks the overflow style and overwrites data if it is *overflow_newest* or *overflow_oldest*; it blocks otherwise.

```
void Read (char* data);
```

Parameter
`data`: pointer to location be read string into.

Description
This method reads a string from the data channel buffer into `data` array. Its source process blocks waiting for data if the data channel is empty.

4.7 Template subclass `Q_Scalar_Data_Channel`

```
Q_Scalar_Data_Channel(int size, enum Overflow_Style os,
bool interrupt);
```

Parameters
`size`: maximum number of data elements to be stored in data channel.
`os`: one of: *blocked*, *overwrite_oldest* and *overwrite_newest*.
`interrupt`: true if the data channel is interrupt-driven and false otherwise.

Description
This is the constructor for queued data channel objects. It creates a queued data channel object of required scalar type by calling the `Data_Channel` base class constructor with element size = sizeof(element type).

```
void Write (Scalar_T data);
```

Parameter

`data`: data element to be written of type `Scalar_T`, where `Scalar_T` is the template parameter for the scalar data channel, which could be any scalar data type.

Description

This method writes the passed data element into the data channel buffer. If the data channel is full, it checks the overflow style and overwrites data if it is *overflow_newest* or *overflow_oldest*; it blocks otherwise.

```
void Read (char* data);
```

Parameter

`data`: pointer to location be read string into.

Description

This method reads a string from the data channel buffer into `data` array. Its source process blocks waiting for data if it is empty.

4.8 Template subclass `Mailbox_Scalar_Data_Channel`

```
Mailbox_Scalar_Data_Channel(enum Overflow_Style os, bool
interrupt);
```

Parameters

`os`: one of: *blocked*, *overwrite_oldest* and *overwrite_newest*.

`interrupt`: true if the data channel is interrupt-driven and false otherwise.

Description

This is the constructor for the mailbox data channel class. It creates a mailbox data channel object of required scalar type by calling the `Data_Channel` base class constructor with size 1 and element size = sizeof(element type).

Other API

Same as Q_Scalar_Data_Channel – section 4.7.

Chapter 5

Kernel Implementation

DARK++ is an embedded kernel implemented in C++ with some components implemented in assembly. It runs on Analog Devices SHARC 21xxx 32-bit digital signal processors. This chapter discusses the major implementation details of the kernel.

It is organized as follows. Section 5.1 discusses the kernel initialization and startup details. Section 5.2 contains a discussion of the implementation of high-speed context switching by exploiting the dual register set hardware. The implementation details of the DARK++ scheduler are explained in section 5.3. In sections 5.4 and 5.5 we explore the ECO implementation details and the data channel-related APIs provided, respectively. Section 5.6 contains details about thread management implementation. Section 5.7 discusses interrupt handling. The last two sections explain the implementation details of time management and real-time support.

Before we start discussing the implementation details, let us take a look at another diagram of the system that includes certain internal data structures used by the DARK++ system. We introduce to the reader, three classes – `ready_queue`, `event_queue` and `waiting_queue`.

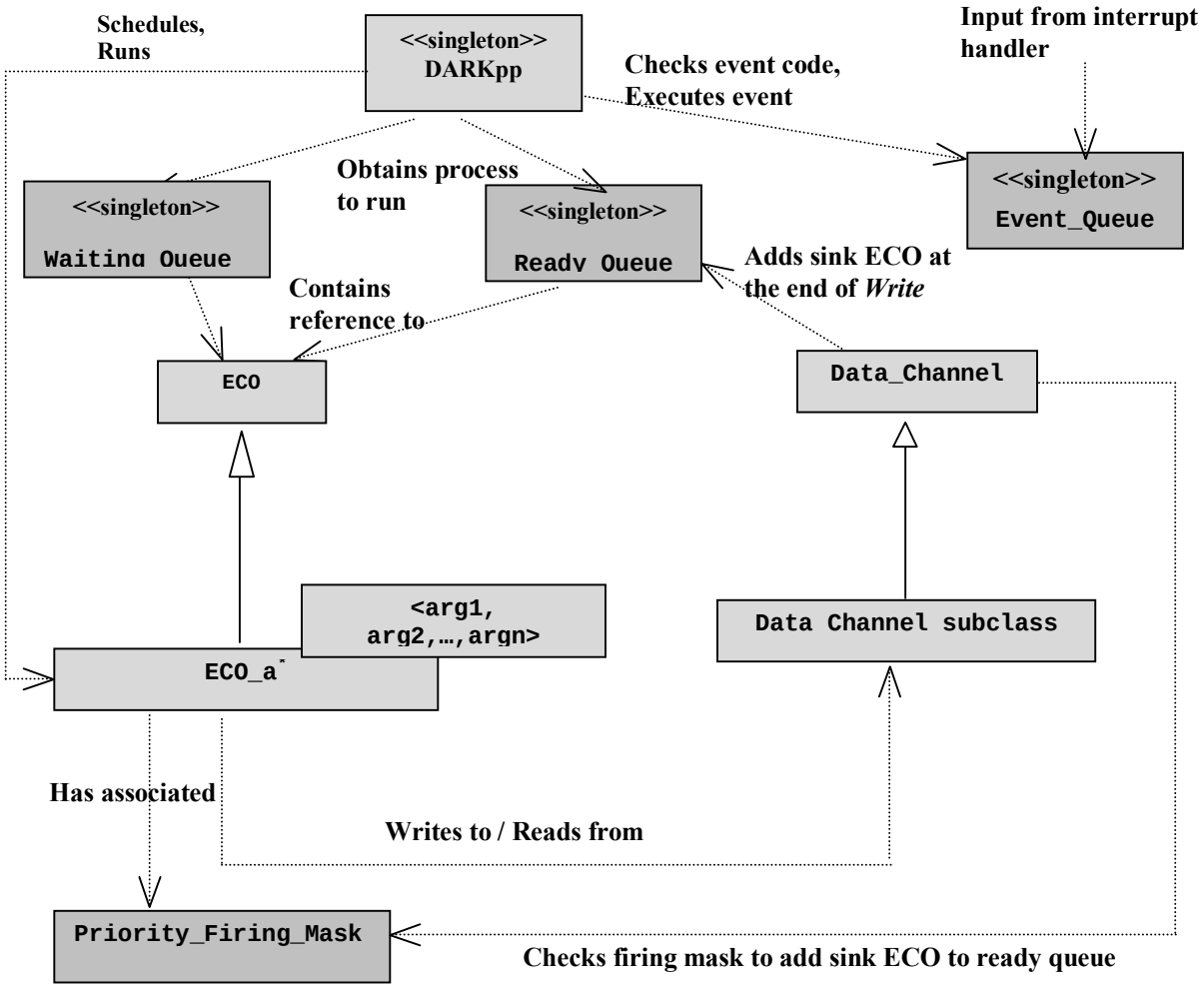
The ready queue is a priority-based heap that stores the ready ECOs. The scheduler uses it to dispatch the next ready process. Every *write* operation to a data channel adds the sink ECO to the ready process if the corresponding ECO has a firing mask set as a consequence of the *write*.

The event queue contains 32-bit event codes that are used for interrupt handling. The kernel checks the code that has been logged and performs the event associated with it.

Finally, the waiting queue contains processes that need to wait for a deadline to elapse before they can be fired. This is also used by the scheduler to run the processes when they are ready.

All these classes are singletons (there is only one object of the class).The following diagram shows these classes and their interaction with the other classes. These internal classes are shaded differently than the classes that are relevant to the application interface.

Figure 5.1. DARK++ System Diagram – with internal classes
* - Sample ECO called ECO_a – the specific ECO name goes here



5.1 Kernel Initialization and Startup

Because dynamic creation and destruction of objects causes severe performance issues, in DARK++ all objects are allocated statically as part of startup. The objects allocated include the data channel and ECO objects and the kernel object, the ready queue object, the waiting queue object and the event queue object, which are created by calls to their respective `getInstance` methods (see Chapter 3) that return a reference to a static object. Data channels created could be either interrupt-driven or non-interrupt-driven; this information is specified during creation of the object. For queued data channels, the required capacity of the data channel is specified. ECO objects are created by specifying their required size in bytes, their initial priority, firing rules and configuration information.

The data channel and ECO objects are declared using template instances that are specified as in figures 5.2 and 5.3. Hence the user has to make the appropriate template instantiations in order to be able to create objects of those required types. For data channels, the data type of the data channel is specified as a parameter, while for ECOs, the types of the input and output data channels (the appropriate template instances) are specified.

While the order of declaration of the ECO objects does not matter for dynamic scheduling, if the kernel does static scheduling, the ECOs are run in the same order as their declarations appear before the kernel starts running.

DARK++ provides four configurable options, as discussed in section 3.5.6. They are:

- Full-featured
- Non-preemptive
- Single-threaded dynamically scheduled
- Single-threaded statically scheduled

These options are specified as preprocessor directives in the `DARKpp_cfg.h` file, which the user can change to get the required option, as shown in figure 5.4.

```
typedef Q_Scalar_Data_Channel<float> Q_Float_Data_Channel;
typedef Q_Scalar_Data_Channel<char> Q_Char_Data_Channel;

typedef Mailbox_Scalar_Data_Channel<bool> Mailbox_Bool_Data_Channel;
typedef Mailbox_Scalar_Data_Channel<int*> Mailbox_Int_Ptr_Data_Channel;
typedef Mailbox_Scalar_Data_Channel<float>
  Mailbox_Float_Data_Channel;
```

Figure 5.2 Data Channel template instantiation examples

```
typedef Abc_Dqo
<  Q_Float_Data_Channel, Q_Float_Data_Channel,
   Q_Float_Data_Channel, Q_Float_Data_Channel,
   Q_Float_Data_Channel, Q_Float_Data_Channel > Abc_Dqo_Q;


typedef Adc_Ia
<  Mailbox_Bool_Data_Channel, Mailbox_Float_Data_Channel >
                                                Adc_Ia_Mailbox;
```

Figure 5.3 ECO template instantiation examples

```
/*------- Kernel Options available to the user -------*/

/* Preemptive and multithreaded DARK++ with firing rules
& dynamic scheduling     */
#define    PREEMPTIVE_MTHREADED    0


/* Non-Preemptive and multithreaded DARK++ with firing rules
& dynamic scheduling */
#define    NONPREEMPTIVE_MTHREADED    1


/* Dynamically scheduled, single threaded DARK++ with firing
rules        */
#define    SINGLETHREAD_DYNSCHD    0


/* Statically scheduled, single threaded DARK++ with no firing
rules        */
#define    SINGLETHREAD_STATSCHD   0

/*-----------------------------------------------------*/
```

Figure 5.4. DARK++ Configurable options to be set by the user – in `DARKpp_cfg.h`

With the full-featured version, the kernel and the process run as separate threads and context switching between them is brought about by calls to the assembly `setjmp` and `longjmp` procedures. This version schedules processes based on their firing rules and priorities by using the ready queue, and it allows preemption. The processes make calls to the scheduler during every `read/write` call to check for a higher or equal-priority ready process. If one is found, the scheduler takes over control and runs the ready process. The check for an equal-priority ready process is performed to ensure fair scheduling.

In non-preemptive scheduling, the kernel schedules processes based on firing rules, but the scheduler does not preempt a running process thread. In other words, processes do not make calls to the scheduler during every data channel operation. Instead every process thread suspends itself, typically when it is waiting for data. The single-threaded versions provide a significant performance boost at the cost of lesser flexibility and fewer features.

5.2 Context Switching

When control is switched from an executing thread to another thread, the state of the executing thread has to be saved and the saved state of the new thread has to be loaded. This process is known as context switching. Context switching, although a necessary part of the entire scheduling process, is an overhead that should preferably be minimized, because no useful work is done during context switching.

Context switching time is often dependent on the support provided by the underlying hardware. There are some processors that provide multiple register sets. In these cases, different register sets can be dedicated to different threads and context switching has to be done only when the number of threads to be run at any time exceeds the number of register sets.

The Analog Devices SHARC 21xxx processors provide two register sets – the primary set and the alternate set. DARK++ exploits these to better the context switching performance. Only selected registers that are used by the C++ run-time environment are saved and restored. Since most context switches in dataflow applications occur between the scheduler and the executing thread, minimizing these context switch times can lead to a significant boost in the overall performance. Hence, DARK++ uses the primary register set for the scheduler thread and the secondary set for the process threads. Thus only some common system registers have to be saved/restored during a context switch and a mode bit has to be set to indicate which of the register sets is to be used when there is a context switch between the scheduler thread and a process thread. It is only when there is a context switch between two process threads that all registers need to be saved/restored.

Custom-written `setjmp` and `longjmp` assembly procedures are used to accomplish context switching. These are described below:

`int setjmp_all(&environment)` – Control can enter this procedure in two ways – either through an explicit call or as a return from `longjmp_all`. When the procedure is entered through a call, it returns 0 and when through a return from `longjmp_all`, it returns the value specified in the call to `longjmp_all`. The `environment` parameter specifies the memory location where *all* C++ runtime registers are to be saved. The procedure saves all the C++ run-time registers and also the instruction pointer at the end.

`void longjmp_all(&environment, return_value)` – This procedure is used in conjunction with `setjmp_all`. It restores all C++ runtime registers and the instruction pointer at the end from the `environment` parameter. Restoring the instruction pointer causes return to `setjmp_all`. The `return_value` parameter specifies the value that `setjmp_all` has to return. There is no return to `longjmp_all`.

`int shallow_setjmp(&environment)`- This procedure is similar to `setjmp_all`, except that instead of saving *all* C++ runtime registers, it saves only the common system registers.

`void shallow_longjmp(&environment, return_value)` – This is used in conjunction with `shallow_setjmp`. It is similar to the `longjmp_all` procedure except that it restores only common registers instead of all registers. It restores the instruction pointer at the end too. Just like in the case of `longjmp_all`, there is no return from this procedure.

Context switches between the scheduler and a process thread use `shallow_setjmp` and `shallow_longjmp`, while context switches between two process threads use the `setjmp_all` and `longjmp_all` procedures.

5.3 Scheduling

Scheduling with the multithreaded versions is performed with the ready queue (heap), which forms a priority queue for ready processes. The class used for this is the `ready_queue` class.

The ready queue stores pointers to ECO processes that are currently in the *ready* state. It is implemented using a heap of pointers to ECO objects and has methods for inserting a node to the heap, deleting the top of the heap and other important heap management methods.

A static `getInstance()` method is used to get the reference to the ready queue object**.**

Scheduling as done in each of the four configurable options has been explained in Section 5.1. The pseudo-code of the scheduler for the four versions is shown in the following figures.

Since in dataflow applications scheduling is determined not only by process priority but also by the presence/absence of data in the input data channel to the process, a major part of scheduling happens as part of the read/write functions.

When a process writes data to a data channel, it sets the `in_ports_ready` variable for the sink process of that data channel appropriately, checks the firing rule of the sink process and if the sink process is found to be eligible to run, inserts it into the ready queue. The sink process acquires a new priority and the `wakeup_call` variable for the sink process is set equal to the firing mask that triggered it.

When a process reads data from a data channel, it resets the firing mask for the source process. Both at the end of read and write, there are macros that check if there is any higher or equal priority ready process. If any are found, control is returned to the scheduler to enable it to run the new process. Figure 5.8 shows the segment of code in the write operation that is responsible for scheduling.

Some other crucial functions involved in scheduling are `wait_to_fire`, which is used to check whether to execute a process another time after it completes one cycle of execution, or to transfer control to the scheduler, `timed_wait_to_fire`, which switches to the scheduler either at the end of one execution of the process or a specified delay, whichever is first, and `delay`, which goes to the scheduler after the specified delay.

```
while(true)
{
   if(deadline_queue_size > 0)
      check the deadlines
   if(actions_pending <> no_actions)
   {
      handle delayed threads
      handle interrupt events
   }
   if(ready_queue_size > 0)
   {
      if(first execution of scheduler OR
                     any process ready with priority >=
                        current_process.priority)
      {
                     current_process = ready_queue.topofheap
         execute current_process
      }
      else if(current_process = ready)
         execute current_process
   }
```

Figure 5.5. Pseudo-code of DARK++ scheduler – for multithreaded versions

```
if(ready_queue.number_of_processes > 0)
{
   current_process = ready_queue.topofheap
   execute current_process
       current_process.state = next_state(current_process.state)
}
```

Figure 5.6 Pseudo-code of DARK++ scheduler – for single-threaded dynamically scheduled version

```
for i=0 to number_of_processes do
   execute process[i]
```

Figure 5.7 Pseudo-code of DARK++ scheduler – for single-threaded statically scheduled version

```
FiringRule temp_rule   = snk_ECO->FIRE_Rule();

snk_ECO->SetIn_Ports_Ready(sink_ECO->In_Ports_Ready() | (1 << sinkport_num));

while (!temp_rule->Last())
{
   if ((snk_ECO->In_Ports_Ready() & temp_rule->Firing_Mask())
         == temp_rule->Firing_Mask())
   {
      snk_ECO->SetWakeup_Call(temp_rule->Firing_Mask());
      snk_ECO->SetProcess_State(ready);
      the_ready_queue.insert(snk_ECO);
      return;
   }
   temp_rule++;
}
```

Figure 5.8. Segment of code in write operation responsible for scheduling

5.4 ECO Implementation

The `ECO` class is an abstract base class for ECOs containing the ECO implementation method as a virtual method. Since ECOs are distinct components with distinct computation/comparison functions, the implementation method is defined for a specific ECO in the subclass corresponding to that particular ECO.

The virtual method is defined in the base ECO class as below:

```
virtual void Implementation()=0;
```

Figure 5.9. Implementation method – declared virtual in class ECO

Every ECO has a firing rule and a priority associated with it. A *firing rule* is a set of *firing masks*, each of which defines one of the possibilities that will cause an ECO to *fire* or begin operation. This means that an ECO can have more than one firing mask associated with it. It would then perform one particular sequence of operations if woken up by one firing mask, and an alternative sequence of operations if fired by another one. These firing masks are based on the status of the ECO's input data channels that is required to cause them to fire. Hence every time after a data token is read from a data channel or written to a data channel, the status of the data channel is checked for *empty* and *non-empty* respectively, and the masks of all sink ECOs of that data channel are updated. The firing mask is represented by a binary mask that specifies the input data channels, which, if filled, should trigger the ECO to fire. For example, the firing mask 00000101 indicates that the ECO is ready to fire when it has data on channels 0 and 2. Every data channel *read* or *write* operation updates the corresponding bit of this mask. The firing masks are arranged in the firing rule by priority. Firing masks are generated using the `Prioirty_Firing_Mask` class. This class also has a `priority` field
associated with a firing mask, which is the new priority that is assigned to the ECO thread if it is triggered as a result of the corresponding firing mask.

The specific implementation method in the ECO subclasses is a loop that repeats until `wait_to_fire` returns false. The `wait_to_fire` method checks if the current process is ready to be fired again and returns true if it is. If not, it swaps to the scheduler thread.

```
template < class Float_Data_Channel_i1, class Float_Data_Channel_i2,
             class Float_Data_Channel_i3, class Float_Data_Channel_i4,
     class Float_Data_Channel_o1, class Float_Data_Channel_o2 >
void Abc_Dqo <class Float_Data_Channel_i1, class Float_Data_Channel_i2,
     class Float_Data_Channel_i3, class Float_Data_Channel_i4,
     class Float_Data_Channel_o1, class Float_Data_Channel_o2 >
:: Implementation()
{  do
   {  switch (wakeup_call)
      {   case ABC_DQO_FIRING_MASK_DEFAULT: default_action();
                    break;
          case ABC_DQO_FIRING_MASK_EXCEPTION: exception_handling();
                    break;
      }
   } while(wait_to_fire());

}
```

Figure 5.10. Implementation method of Abc-Dqo ECO

In every iteration, the `wakeup_call` data member is checked to find which firing mask triggered the ECO, and the method corresponding to that firing mask is called. Figure 5.10 shows the implementation method of an Abc-Dqo converter.

5.5 Data Channel Operations

The read and write operations have different implementations for the queued and mailbox data channels. While these operations are implemented as macros in the C version of the kernel, in DARK++ these are methods and hence there is some performance overhead. The implementation of the read/write operations is different for byte data channels, string data channels and other data channels used for storing scalar data types. The read and write operations for byte data channels are present in the base `Data_Channel` class.

The read and write methods make calls to the `block_on_read`, `read_goto_os`, `block_on_write` and `write_goto_os` macros. Macro `block_on_read` causes the data channel to be blocked waiting for data. `block_on_write` causes a full data channel to be blocked if its overwrite style is *blocked*. On the other hand, if the data channel has an overwrite style is *overwrite_oldest* or *overwrite_newest*, the channel is overwritten with the incoming data, replacing the oldest/newest data item, respectively. The `read_goto_os` and `write_goto_os` macros are called at the end of the read and write operations respectively, and check for a higher or equal priority ready process in the ready queue, and if there is, cause a context switch to the scheduler.

```
inline void Write(Scalar_T in_buf)
{
       unsigned int i; //index
   int temp_rear;
   unsigned int rem_cap_buffer;

   if(element_size <= size)
   {
      BLOCK_ON_WRITE_QUEUED(this, element_size);
      *(Scalar_T*) (buffer + rear ) = in_buf;
      rear = ( rear + element_size );
      if ( rear == size)
         rear = 0;
      num_entries++;
      setmask_on_write();

         // switch to OS once
      WRITE_GOTO_OS(interrupt_driven);
   }
}
```

Figure 5.11. Implementation of Write operation for Queued data channel

```
inline void Write(Scalar_T in_buf)
{
   BLOCK_ON_WRITE_MAILBOX(this);
   *(Scalar_T*) (buffer + rear ) = in_buf;
   num_entries = 1;
   setmask_on_write();
   // switch to OS once
   WRITE_GOTO_OS(interrupt_driven);
}
```

Figure 5.12. Implementation of Write operation for Mailbox data channel

Figures 5.11 and 5.12 show the implementation of the write operation for a queued data channel and for a mailbox data channel, respectively. Figure 5.13 shows the implementation of the `block_on_write` macro, which handles data overflow in a data channel.

```
  #define BLOCK_ON_WRITE( data_channel, length )\
    { \
      while(full(data_channel))\
      { \
            switch (data_channel->overflow_style) \
            { \
                case OS_Block  :  /*block*/ \
                                  data_channel->setBlocked(); \
                                  Current->swap( blocked ); \
                                  break; \
                case OS_Overwrite_Newest: /*re-adjust rear*/ \
                                       data_channel->rear -= length; \
                                       if (data_channel->rear < 0) \
                                       { \
                                          data_channel->rear += size; \
                                       } \
                                       break; \
               case OS_Overwrite_Oldest: \
                                       data_channel->front += length; \
                                       if(data_channel->front == size) \
                                       { \
                                          data_channel->front = 0; \
                                       } \
                                       break; \
          } \
        }\
    }
#else
    #define BLOCK_ON_WRITE(data_channel, length)
```

Figure 5.13. Handling data overflow when attempting a Write - implementation of the block_on_write macro

5.6 Thread Management

An ECO can be viewed as a process that executes its `Implementation` code provided by the ECO designer. The various possible states of these processes are: *ready*, *run, blocked*, *wait_for_fire*, *timed_wait*, *timed_wait_for_fire* and *dead*. When the kernel starts, each thread is in the *wait_for_fire* state. A process is in *ready* state once its required input data channels have data tokens in them. A process in the *ready* state is present in the ready queue. The kernel removes the ready process of the highest priority (present at the head of the ready queue) and executes it, changing the state of that process to the *run* state. A process is *blocked* when it tries to write to a full data channel.

After every *write* operation, the mask of its sink ECO (ECO that reads from this data channel) is updated; i.e., the bit corresponding to the data channel in question is set. Thus, while a *read* operation could unblock a process blocked on a data-channel, a *write* operation could fire it.

The `wait_to_fire` function can be used to fire the ECO again. If the ECO is not ready for firing, it goes into the *wait_for_fire* state. The user can also delay the execution of the ECO for a

pre-determined time, which puts the ECO into *timed_wait* state. The *timed_wait_for_fire* state is a combination of *wait_for_fire* and *timed_wait*. An ECO in this state can be fired if a firing mask becomes true *or* if the time period elapses. The ECO goes into the *dead* state once it finishes execution.

Figures 5.14 and 5.15 show the implementation of the `wait_to_fire` and `swap` functions, respectively.

```
inline bool Wait_To_Fire()
{
       FiringRule temp_rule   = F_Rule;
       while (!temp_rule->Last())
   {
      if (in_ports_ready == temp_rule->Firing_Mask() )
        return true;
      temp_rule++;
   }
   swap(wait_for_fire);
   return true;
}
```

Figure 5.14. Implementation of Wait_To_Fire method

```
inline void  swap(ProcessState next_state)
{
   Current->SetProcess_State(next_state);  // Can be other state also
like waiting

   if (next_state == dead)
      shallow_longjmp(the_kernel.OS_Env(),1);

   if (setjmp_all(&(ECOenv)) == 0)
      shallow_longjmp(the_kernel.OS_Env(),1);
}
```

Figure 5.15. Implementation of swap method

5.7 Interrupt Handling

There are many RTOSes that support interrupt handling through the use of compiler-provided mechanisms, using C functions that can be used as interrupt routines. This method involves a substantial overhead in context switching, since all registers are saved
and restored while handling interrupts. The C compiler provided by Analog Devices for its SHARC DSPs supports this approach, and in addition, also provides the option of using the alternate register set for interrupt handling (since the C runtime uses only the primary register set). However, DARK++ cannot use this option, since it uses both the alternate and primary register sets, as explained in Section 5.2.

DARK++ uses an alternative approach for handling external interrupts. This method provides performance comparable to that of using the alternate register set for interrupt handling. Here, rather than placing actions directly in the interrupt handler itself, DARK++ uses a minimal footprint handler that simply logs incoming events into the *event queue*, which is managed by the DARK++ scheduler. The interrupt handler runs in the currently active register set and only

needs to save and restore a couple of registers. It logs a 32-bit code representing the interrupt that was received, into an event queue (class `Event_Queue`) and then returns control to the kernel. The status of the event queue is reflected by the `actions_pending` variable that we have already explained.

The event queue is a circular queue of 32-bit event codes of those events that have caused an interrupt to occur. This class consists of a circular queue and the necessary queue arithmetic operations.

DARK++ also supports clock interrupts and non-maskable interrupts (NMI). The clock interrupt ISR is written in assembly and simply increments the kernel data member, `current_time` that is used for time management. Only a few registers required for incrementing a variable are saved and restored in this ISR. NMI is used for emergency condition notification and requires a time critical response. In most cases, it results in a call to the application's emergency shutdown procedure, bypassing all other kernel as well as application code.

5.8 Time Management

DARK++ provides APIs to allow ECOs to request a timed delay. In most other RTOSes, the kernel checks each waiting thread at every clock tick, and adds it to the ready queue when the waiting period has expired. However, this technique can introduce unnecessary overhead if there are a number of waiting threads. Hence DARK++ uses a different approach to handle timed delays. When the `timed_wait_for_fire()` method is called, the delay is converted into an absolute time by adding the current system time to it and then stored in the ECO (process) object. The thread is then added to the waiting queue in which the threads are arranged in ascending order by absolute time and `actions_pending` is set to `future_actions`.

The kernel checks for `actions_pending` and adds the process back to the ready queue when the deadline has expired. To check whether the deadline has been reached, it compares the system time with the thread wakeup time of the first thread in the waiting queue. The scheduler needs to check only the first thread in the waiting queue unless that thread's waiting period has elapsed.

```
bool ECO ::  timed_wait_to_fire(int delay_time)
 {   FiringRule temp_rule   = F_Rule;
  while (!temp_rule->Last())
      {
     if (in_ports_ready == temp_rule->Firing_Mask())
        return true;
     temp_rule++;
      }
  wakeup_time = delay_time + current_time;
     actions_pending = future_actions;
     the_waiting_queue.insert(this);
     swap(timed_wait_for_fire);
     return true;
 }
```

Figure 5.16. Implementation of the timed_wait_to_fire method in class ECO

```
bool ECO :: delay(int delay_time)
 {
    wakeup_time = delay_time + current_time;
    actions_pending = future_actions;
    the_waiting_queue.insert(this);
    swap(timed_wait);
    return true;
 }
```

Figure 5.17. Implementation of the delay method in class ECO

5.9 Real-time Support

DARK++ provides the user the option of enabling real-time support. It has provision for the user to assign a function handle that will be used to run the provided scheduling algorithm. If this handle is null then the default algorithm that DARK++ uses is the fixed-priority RMA algorithm. In order to meet the high-performance objective, the complex real-time support necessary for POSIX compliance has been avoided in DARK++.

The RMA algorithm works as follows. A set of N tasks can be scheduled using rate-monotonic priorities if and only if [13, 32]:

$$\sum_{i=1}^{N} C_i/T_i \leq U(N)$$

where C_i: execution time for task i and

T_i: its period

The ratio C_i/T_i is the CPU utilization of task i.

U(N) is the processor utilization for N tasks

U(N) is given by the following equation [13, 32]:

$$U(N) = N(2^{1/N} - 1)$$

The equation converges to 0.693 for large values of N. So it may be concluded that a set of N tasks is schedulable using RMA if the sum of their CPU utilization factors is <= 0.693.

DARK++ uses this algorithm to determine whether a given set of threads is schedulable before assigning priorities. If scheduling is feasible, then the shortest thread is assigned maximum priority.

DARK++ provides the following simple API to monitor real-time deadlines.

A deadline for an ECO can be set using the method:

```
void ECO :: set_deadline(int time);
```

The `time` parameter specifies the time by which the ECO has to finish its execution. When this method is invoked, `time` is converted to an absolute time by adding the current system time to it and the ECO is added to the *deadline queue*.

The following method can be used to ascertain whether an ECO has met its deadline:

```
bool ECO :: check_deadline();
```

This method removes the ECO from the deadline queue and returns true if the ECO met its deadline.

The DARK++ scheduler checks the first entry of the deadline queue in each switching cycle. If it finds an ECO that missed its deadline, it calls a user-provided handler.

Chapter 6

Experimental Evaluation

This chapter explains the performance experiments carried out with DARK++. It presents the performance results and a comparison with the corresponding results for DARK. We also explain the reasons and the implications of the obtained results. We have carried out empirical evaluation using three power control applications – the open-loop 3-phase inverter, the closed-loop 3-phase inverter and the closed-loop control for 3-phase boost rectifier.

In the following section we explain dataflow applications and the three applications used in our evaluation in particular, providing a brief overview of the various ECOs in each of them and an explanation of the primary functions performed in a switching cycle. We present the dataflow graph for each of them. Following this, in section 6.2 we present the results of the experiments and we conclude with a discussion of the obtained results and evaluation and comparison of the OO kernel with its non-OO counterpart.

6.1 Dataflow Applications

Dataflow software comprises a number of independent and standard components communicating through data channels. Here we call the components ECOs (Elementary Control Objects). A library of ECOs exists and the code is reused in all applications that need to make use of any of the ECOs.

ECOs [20] are of three types – computational ECOs, coordination ECOs and driver ECOs. A computational ECO performs some required computation that forms input to the next ECO. A coordination ECO is used to support transparent control and coordination of distributed hardware components. A driver ECO handles hardware dependencies and forms a standard interface to control hardware.

The following subsections explain three power-electronic dataflow applications that have been used to carry out performance experiments. These are – open-loop 3-phase inverter, closed-loop 3-phase inverter and boost rectifier. Following these, Section 6.2 presents the empirical evaluation details and the results.

6.1.1 Open-loop 3-phase Inverter

The DFG (dataflow graph) for the open-loop 3-phase inverter application [30,31] is shown in figure 6.1.

This is the simplest application that we have used in our experiments. It consists of three *Lookup_Sin* ECOs that receive a *Start* signal from their boolean input data channels. They look

up a value from a circular table that they maintain using a table pointer. After every look-up, the table pointer is incremented. The table source and the modification step for the table pointer are stored as part of the ECO's configuration information. The output values of the three ECOs have a phase difference of 120 degrees. These in combination, form input to the *Modulator* and fire it to produce three floating-point results that form inputs to the three *PEBB_drivers*. The *PEBB_drivers* convert the data from floating-point format to a format of control information that can be understood by the power stage, which they will be inputs to.

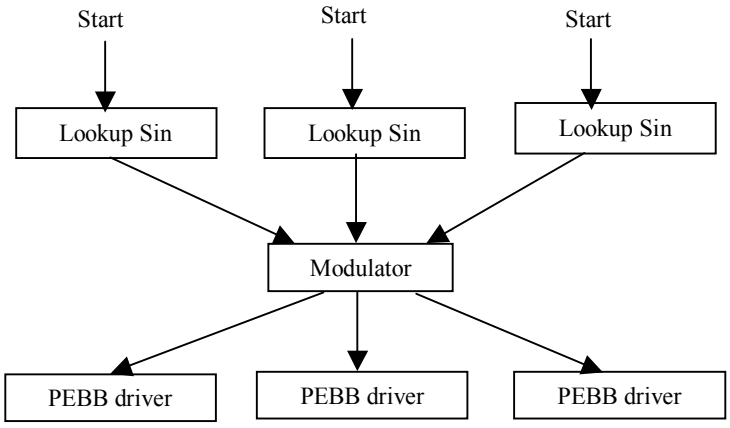


Figure 6.1. Open-loop three-phase inverter

The following two applications are relatively more sophisticated applications than the open-loop application.

6.1.2 Closed-loop 3-phase inverter

A switching cycle in the closed-loop 3-phase inverter [31] starts with the firing of the two *Lookup_Sin* ECOs by external interrupts, and the input of external feedbacks from A/D converters to the *Abc_Dqo* ECO.

The outputs of the *Lookup_Sin* ECOs go to 1-2 duplicators because the same output has to be duplicated for both the *Abc_Dqo* and the *Modulator*. On receiving both the feedback information and the outputs from the *Lookup_Sin* ECOs, the *Abc_Dqo* ECO transforms the input abc coordinates to dqo coordinates. These dqo coordinates form inputs to the 2-d regulator, which performs PI regulation to get duty cycles in the dqo coordinates. The *dqo_albe* ECO transforms the dqo coordinates back to αβγ. After this, the 3-d modulator performs modulation on the duty cycles. The generated duty cycle information forms input to the power stage.

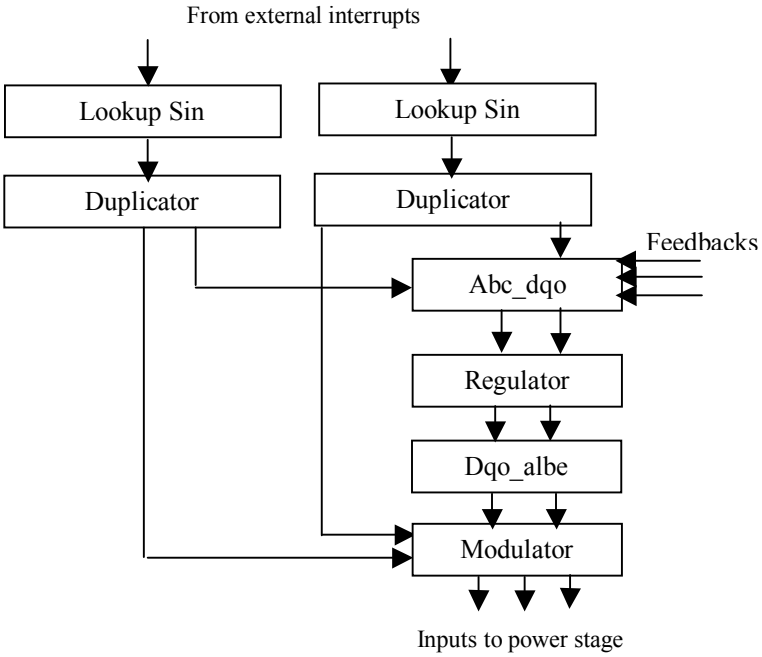


Figure 6.2. Closed-loop three-phase inverter

6.1.3 Boost Rectifier

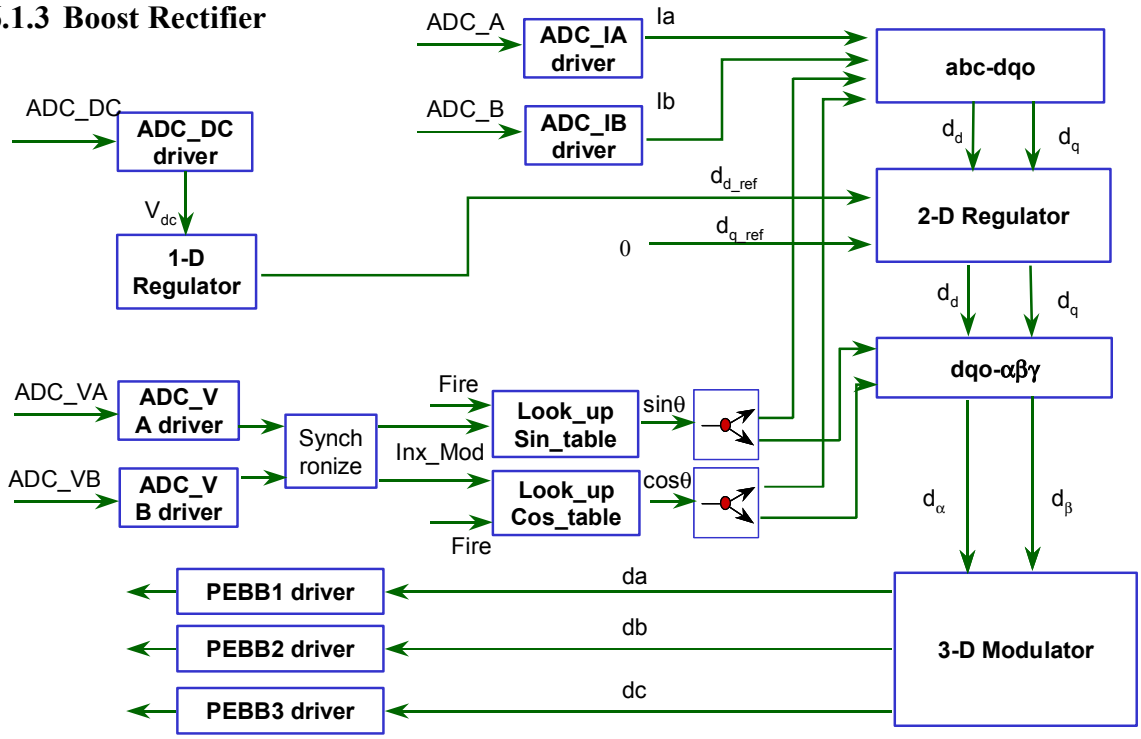


Figure 6.3. Closed loop control for 3-phase boost rectifier

This is a dataflow graph for a 3-phase boost rectifier [20] closed loop control. There are two control loops: current loop and voltage loop. The voltage loop should be executed first to generate reference for the current loop. For the 3-phase current loop control, the dq transformation technique is used.

All sensed data are implemented as interrupt-driven data channels, synchronized by the switching clock. The rising edge of the switching clock causes the execution of the corresponding interrupt handler, in which all the sensed data, phase currents and voltages

and dc voltage are updated. Those ADC drivers then translate those sensed data to correct values, respectively. At this point ECO *1-D regulator* and *synchronize* are ready. The dc voltage is regulated in the ECO *1-D regulator*, and the current loop reference d_{d_ref} is generated. The *synchronize* ECO tested the phase voltages and generate a boolean output to indicate whether the phase angles need synchronizing.

The two *lookup_table* ECOs have two different behaviors depending on different inputs combinations. At normal condition, if the phase voltages do not need synchronizing, the *lookup_table* ECOs increment their table pointers and output the table value; otherwise, the table pointers will be reset. Through ECO *duplicator*, the sin and cos values are copied and directed to two different ECOs. So far, the *abc_dqo* ECO is ready to transform the phase currents in abc coordinates to dqo coordinates. Then the *2-D regulator* performs the current loop regulation with the reference generated from the voltage loop. The regulated currents in dqo coordinates will then be transformed back in αβγ coordinates through ECO *dqo-αβγ*. Then the ECO *3-D Modulator* is ready to synthesize duty cycle information for each phase. In the *PEBB_driver* ECO, the duty cycle information will be translated to the form that can be used to generate switch pulse at the phase leg.

6.2 Performance Results

This section discusses the results obtained during the performance evaluation experiments of the kernel. The experiments were conducted on Analog Devices-SHARC 21160 digital signal processor. The Analog Devices VisualDSP++ simulator was used to run the experiments and collect profiling information. Experiments were conducted on the three dataflow applications described.

We present the results obtained by comparing the performance of the different versions of DARK++ and DARK (full-featured, non-preemptive, single-threaded statically scheduled, single-threaded dynamically scheduled) on these three control applications. Tables 6.1 through 6.3 show the data for the three applications run using message queues. Tables 6.4 though 6.6 show the data for the applications when run using mailboxes. The tables show the total number of instruction cycles taken by DARK and DARK ++ for one switching period of the kernel for the mentioned applications. This is broken down into six categories of operations– ECO execution, dispatcher, context switching, ready queue operations and other operations. We can compare the contributions of each of these factors to the execution time for the application in

the case of DARK++ with those in the case of DARK. Following this, we present graphs that are obtained by normalizing the overhead of DARK++ over the three control applications. Figure 6.4 is based on the performance of the applications using queued data channels and figure 6.5 shows the same data for the applications run with mailboxes.

Table 6.1. Performance Results in terms of number of instruction cycles for the open-loop inverter– with message queue data channels

Operations	Full-featured		Non-preemptive		Single-threaded dynamic scheduled		Single-threaded static scheduled	
	DARK	DARK++	DARK	DARK++	DARK	DARK++	DARK	DARK++
ECO execution	235	186	235	186	235	186	235	186
Dispatcher	529	425	529	425	219	177	212	192
Context switching	1148	1148	1148	1148	0	0	0	0
Ready queue operations	525	924	525	882	525	831	0	0
Data channel operations	1304	917	999	680	945	680	312	536
Other OS operations	462	626	460	486	77	280	0	0
Total	**4203**	**4226**	**3896**	**3807**	**2001**	**2154**	**759**	**914**

Table 6.2. Performance Results in terms of number of instruction cycles for the closed-loop inverter– with message queue data channels

Operations	Full-featured		Non-preemptive		Single-threaded dynamic scheduled		Single-threaded static scheduled	
	DARK	DARK++	DARK	DARK++	DARK	DARK++	DARK	DARK++
ECO execution	583	620	583	623	583	623	583	623
Dispatcher	679	553	679	553	297	270	270	268
Context switching	1476	1476	1476	1476	0	0	0	0
Ready queue operations	612	1719	612	1236	612	1031	0	0
Data channel operations	2659	2226	2059	1526	2059	1526	731	926
Other OS operations	594	881	594	727	203	467	0	0
Total	**6603**	**7475**	**6003**	**6141**	**3754**	**3917**	**1584**	**1817**

Table 6.3. Performance Results in terms of number of instruction cycles for the boost rectifier– with message queue data channels

Operations	Full-featured		Non-preemptive		Single-threaded dynamic scheduled		Single-threaded static scheduled	
	DARK	DARK++	DARK	DARK++	DARK	DARK++	DARK	DARK++
ECO execution	778	638	778	638	778	638	778	638
Dispatcher	1354	1057	1354	1057	585	495	567	529
Context switching	2952	2952	2952	2952	0	0	0	0
Ready queue operations	1811	2240	1811	2190	1811	2028	0	0
Data channel operations	4252	3762	3174	2439	3174	2439	1020	1852
Other OS operations	1224	2180	1224	1844	205	1317	0	0
Total	**12371**	**12829**	**11293**	**11120**	**6553**	**6917**	**2365**	**3019**

The execution time for the three applications, as we may note from the above tables, increases as the applications increase in complexity, with more number of ECOs and therefore, more computation, more communications, and also increased context switching and scheduling overhead. We see that the full-featured version involves maximum execution time since it provides the maximum number of features; it performs a check for an equal or higher priority ready process at the end of every *Read* and every *Write* operation and transfers control to the kernel if there is one. With the non-preemptive version of the kernel, an executing thread necessarily has to run to completion before another thread can begin execution, even if a higher priority thread becomes ready during the execution of the current thread. Hence process threads need to do no checking and switching of control to the scheduler. This significantly brings down the execution time. The single-threaded versions have no notion of separate process and scheduler threads. Instead, every process is run by a normal method call. Therefore single-threaded systems are necessarily non-preemptive. The single-threaded dynamically scheduled version supports the notion of firing rules and processes are scheduled dynamically based on the sequence of *Writes* and the corresponding triggers to the sink processes. Since the process to be run at any time is determined dynamically, this version of the kernel still involves ready queue management operations. Hence although it is slower than the earlier two versions discussed, it is slower than the single-threaded version in which processes have a pre-assigned execution order and the kernel essentially is a dispatcher and does no scheduling.

We now present the performance results for the three applications run with mailbox data channels. Mailboxes are data channels with unit capacity. Hence mailbox data channel management is much simpler involving no queue arithmetic, and simpler overflow handling.

Table 6.4. Performance Results in terms of number of instruction cycles for the open-loop inverter – with mailbox data channels

Operations	Full-featured		Non-preemptive		Single-threaded dynamic scheduled		Single-threaded static scheduled	
	DARK	DARK++	DARK	DARK++	DARK	DARK++	DARK	DARK++
ECO execution	235	186	235	186	235	186	235	186
Dispatcher	529	425	529	425	219	177	212	192
Context switching	1148	1148	1148	1148	0	0	0	0
Ready queue operations	525	924	525	882	525	831	0	0
Data channel operations	1040	521	734	307	734	307	67	280
Other OS operations	462	626	450	486	121	357	0	0
Total	**3939**	**3830**	**3621**	**3434**	**1834**	**1858**	**514**	**658**

Table 6.5. Performance Results in terms of number of instruction cycles for the closed-loop inverter– with mailbox data channels

Operations	Full-featured		Non-preemptive		Single-threaded dynamic scheduled		Single-threaded static scheduled	
	DARK	DARK++	DARK	DARK++	DARK	DARK++	DARK	DARK++
ECO execution	778	638	778	638	778	638	778	638
Dispatcher	1354	1057	1354	1057	585	495	567	529
Context switching	2952	2952	2952	2952	0	0	0	0
Ready queue operations	1811	2240	1811	2190	1811	2028	0	0
Data channel operations	3699	2071	2640	1588	2640	1575	216	930
Other OS operations	1224	2180	1224	1844	88	1317	0	0
Total	**11818**	**11138**	**10759**	**10269**	**5902**	**6053**	**1561**	**2097**

Table 6.6. Performance Results in terms of number of instruction cycles for the boost rectifier – with mailbox data channels

Operations	Full-featured		Non-preemptive		Single-threaded dynamic scheduled		Single-threaded static scheduled	
	DARK	DARK++	DARK	DARK++	DARK	DARK++	DARK	DARK++
ECO execution	583	620	583	623	583	623	583	623
Dispatcher	679	553	679	553	297	270	270	268
Context switching	1476	1476	1476	1476	0	0	0	0
Ready queue operations	612	1719	612	1236	612	1031	0	0
Data channel operations	2271	1041	1698	887	1698	888	220	333
Other OS operations	594	881	594	727	181	467	0	0
Total	**6215**	**6290**	**5642**	**5502**	**3371**	**3279**	**1073**	**1224**

The above results indicate that DARK++ has performance comparable to that of DARK. The full-featured version of the kernel, in particular, outperforms that of DARK for the open-loop inverter and for the boost rectifier applications running on mailbox data channels, while the closed loop inverter running with mailbox data channels on DARK++ is a shade slower than on DARK (1.2% slower).

Following are graphs that provide a good summary of all of the above data. They present the overheads imposed by each of the four versions of the two kernels, normalized over the three control applications. The first graph is for the applications run using message queues and the second one is for the applications run using mailboxes. After presenting these graphs, we will discuss the results gathered by comparing the OO versus the non-OO kernel.

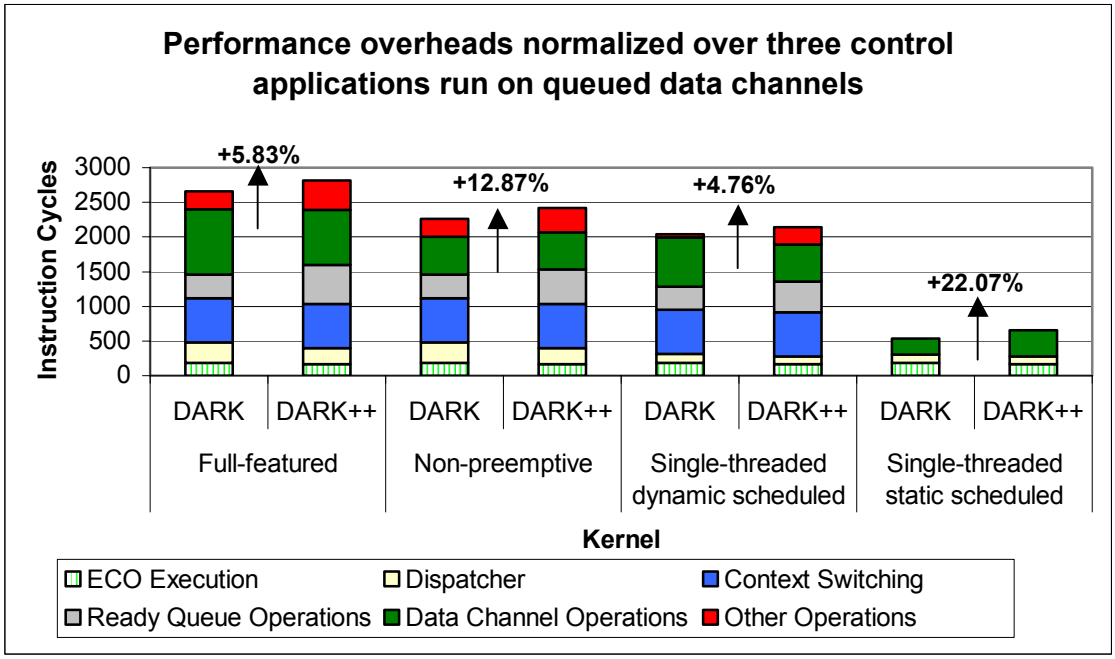


Figure 6.4. Performance results for the two kernels with message queues

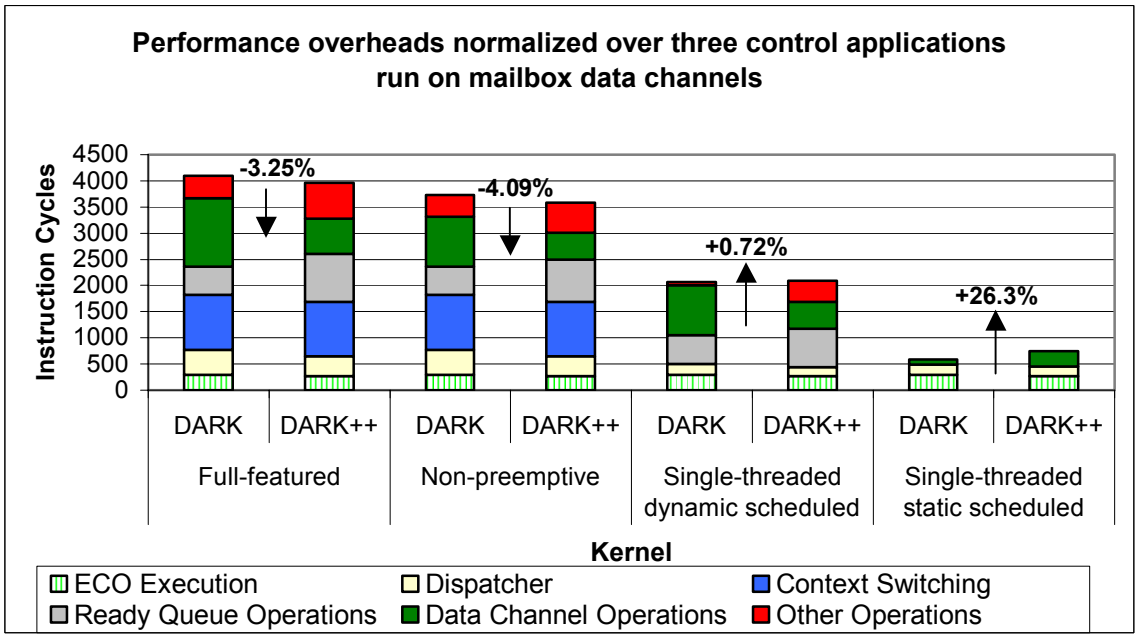


Figure 6.5. Performance results for the two kernels with mailboxes

6.3 Discussion of the performance data

It may be observed that the multithreaded versions of DARK++ running the applications using message queues resulted in marginally lesser speed than that of DARK, while running the

applications using mailboxes showed a performance gain. The data channels operations, which are implemented as macros in the C version of the kernel, are inlined
methods in the C++ versions. Because these operations were actually being inlined by the compiler more often in the case of mailboxes than in the case of message queues, there was a significant performance gain.

The single-threaded versions of DARK++ impose significant overhead. The DARK++ dispatcher for this version is slower than the DARK dispatcher unlike for the other versions. This is because, while the multithreaded kernel uses calls to *setjmp* to execute processes in all but the first switching cycle of the kernel, the single-threaded kernel always makes an explicit call to the ECO `Implementation` method and since this is a virtual method, there is a considerable overhead introduced due to the dynamic resolution to effect the call.

The context switching times are equal in DARK and DARK++ since both use the same custom *setjmp* and *longjmp* assembly functions to accomplish this. While the ECO execution and the scheduler execution times are comparable in the two cases, the ready queue operations have a significantly higher contribution to the entire execution time in the case of DARK++ than in DARK. This is because all accesses to any of the ready queue data members have been counted under this category and there are a number of calls to such operations – e.g., calls to a method that returns the number of items present in the ready queue. Such calls are made both by the scheduler, as well as from the data channel operations in the case of preemptive scheduling, to check for other high-priority ready processes that may be waiting. Typically, it has been observed that simple methods that return the value of a data member take between 7-9 instruction cycles. Hence even if such a method is inlined by the compiler, there is an overhead incurred by the frequent use of such methods.

The “other operations” category also takes more number of instruction cycles in the case of DARK++ than in DARK due to the same reason as mentioned above. There are some generic methods that are frequently used by various callers to retrieve some data members and these introduce significant overhead.

It is worth noting that a great many operations that are specified as macros in DARK are class methods in DARK++, with the “inline” keyword. Therefore, while these operations are guaranteed to be preprocessed and efficient in DARK, many of them are not inlined by the compiler in DARK++. A better performance could have been achieved with DARK++ if it were possible to guarantee inlining of all methods. Also, the unpredictability of a method actually being inlined could lead to marginal irregularities in performance.

The single-threaded versions of the kernel need do no context switching and hence the number of instruction cycles for the category is zero. The single-threaded statically scheduled kernel makes use of a precomputed order to execute the processes sequentially and hence does not use the ready queue.

6.4 Summary

From the data gathered on these three applications and from the above discussion, we may conclude that careful design in OO paradigm can yield appreciable performance. We summarize below, the most important points about OO design and performance issues:

- As we have seen, it very naturally imposes the need for more method calls. While one can choose to specify such methods with the "inline" keyword, since it relies on the discretion of the compiler, there may be inefficiencies (if the compiler does not actually inline them). The disadvantage with inlining is that for huge applications, the entire code may not fit into memory if the memory offered by the embedded system hardware is limited.

- Another related point is that, while it is often worthwhile to specify some frequently used (small) operations as macros in C, it may be inappropriate to do this in C++ (an OO language) where more often than not, we want operations as methods in a class and specifying these as macros might lead to a sloppy design. In DARK++, as stated earlier, we have specified a few generic operations used by the data channel *Read* and *Write* methods as macros. The question really is a tradeoff between elegance and performance.

- It is best to avoid virtual methods as these rely on dynamic binding, which impact performance considerably. The performance numbers for the single-threaded statically scheduled version of DARK++ reflect this fact very clearly. However, if the system being designed compels the use of virtual methods, one necessarily pays for the V-table look-up and resolution during run-time. However, we often lose the flexibility and natural extendibility through inheritance when we avoid usage of virtual methods.

- There are some important points to remember while working on OO design for performance-critical systems. Use of dynamic memory allocation, perhaps by creating objects "on the fly" is a bad idea for a system where performance is critical. This should be avoided.

- Templates are often handy and neat to use in the OO design and user-defined templates do not have any inherent performance concerns associated with them since template instantiations take place before run-time.

Chapter 7

Conclusions and Future Work

Our work has shown that with efficient OO design and implementation, it is possible to achieve performance comparable to non-OO implementations. From the standpoint of the classically cited advantages of OO paradigm - modularity, reusability and maintainability, our system is comparable to the non-OO implementation but does not provide any significant improvement since the non-OO implementation has a well-defined structure as well, with a library of reusable ECO components. However, some important advantages of using C++ are compiler-enforced encapsulation and protection, with the notion of classes with data members and behaviors of various protection levels. The most important point in this context is the static type checking that is provided by the C++ compiler that makes the usage of typed data channels much safer than in the C version. For example, the configuration parameters for ECOs have their specific structure defined within the subclass for that particular ECO. So in the constructor for the ECO subclass, one needs to pass the configuration parameter in the right format, failing which compilation does not go through. However, in the C version, all ECOs are treated in a uniform way and are just functions. A character pointer points to the configuration information, increasing the scope for errors. Similarly, the data channels in DARK++ have a well-defined hierarchy through inheritance, with a base data channel class from which template classes that take a particular scalar/user-defined type as a parameter are inherited. As opposed to this, in DARK typed data channels are implemented merely by typecasting to the required data type – this is clearly less type-safe than what C++ offers.

The DARK++ system also offers better extendibility than does DARK. While in DARK, ECOs are written as functions, in DARK++ these are written as classes. Currently, we have a base class for an ECO that has a pure virtual function for the implementation code and this function can be written appropriately for any particular ECO. However the use of the OO approach is not limited to this. If there are ECOs that have certain common functionality, then we can have a common class from which they inherit. Development of one ECO from another is also possible as and when required.

Another significant advantage is templates, which combine the advantage of offering encapsulation and stronger type checking with high efficiency comparable to macros (which are relatively less type safe) if the methods are inlined.

The DARK++ system also offers a better and more structured way of extending data channels for user-defined data types. Although DARK has capability for this, it is weaker since all user-defined data types get treated as a raw stream of bytes in the *Read* and *Write* operations. In DARK++, the user-defined data type is specified as the template parameter, thereby making it more robust and also more structured.

Future Work

A transparent distributed communications protocol has been developed that runs on DARK. This handles communications for dataflow applications with components on multiple processors. This can be added to DARK++ as well.

The DARK++ system takes various declared ECO and data channel objects that form part of the dataflow graph for an application as input components. A graphical user interface can be added to this system that will enable application designers to drag and drop these items to compose the required dataflow graph. This would be a convenient and more natural way of drawing a dataflow graph.

References

[1] "Designing and Implementing Choices: An Object-oriented System in C++" – Roy H Campbell, Nayeem Islam, David Raila and Peter Madany.

[2] "Principles of Object-Oriented Operating System Design" – Roy H Campbell, Gary M Johnston, Peter W Madany, Vincent F Russo.

[3] "Using Naming Strategies to make Massively Parallel Systems work" – Henning Schmidt.

[4] "Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages" – Svend Frolund.

[5] "ACT++ 2.0: a CLASS library for Concurrent Programming in C++ Using Actors" – Dennis Kafura, Manibrata Mukherji, Greg Lavender.

[6] "Object-Oriented Operating System" – Chutima Boonthum - http://www.cs.odu.edu/~cboont/cs871/ooos/OOOSFinalPaper.doc.

[7] "Performance Analysis of Real-time Embedded Software" – Yau-Tsun Steven Li, Sharad Malik.

[8] "DIRECT: Towards a Distributed Object-Oriented Real-Time Control System" – M. Gergeleit, J. Kaiser, H. Streich.

[9] "True Real-Time Embedded Systems Engineering"- Ken Tindell.

[10] "OOPic Programmer's Guide" – Chapter 6- "Introduction to OOP" – http://www.oopic.com/pgchap6.htm.

[11] http://www.engr.csufresno.edu/Personal/CSci/Students/Grad/Rajeshwari_SusaiMicheal/ia.html#Anal

[12] http://pauillac.inria.fr/~remy/work/ojoin/ojoin004.html.

[13] "Real-time Systems design and analysis", Second Edition – Phillip A. Laplante..

[14] http://www-ec.njit.edu/~mxl5294/.

[15] "Advances in the dataflow computational model" – Walid A. Najjar, Edward A. Lee, Guang R. Gao..

[16] "Multithreaded architectures: principles, projects and issues", ACAPS Technical Memo 29, School of Computer Science, McGill University, Montreal, Que., February 1994, ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos – J.B. Dennis and G.R. Gao.

[17] J.T. Buck and E.A. Lee, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model", Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. I, Minneapolis, MN, April 1993, pp. 429-432.

[18] J.T. Buck, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model", Technical Report UCB/ERL 93/69, Ph.D. Dissertation, Department of EECS, University of California, Berkeley, CA 94720, 1993.

[19] E.A. Lee and D.G. Messerschmitt, "Static scheduling of synchronous dataflow programs for digital signal processing", IEEE Trans. Comput. (1987).

[20] J. Guo, S. Edwards, and D. Boroyevich. "Elementary control objects: Toward a dataflow architecture for power electronics control software." In *Proc. IEEE 33rd Annual Power Electronics Specialists Conf. (PESC 2001)*, June 2002.

[21] Jinghong Guo, Stephen Edwards, Dushan Boroyevich, "Improved architecture of PEBB plug and play power electronics systems: Elementary control object (ECO) and dataflow", *CPES 2001 Power Electronics Seminar and NSF/Industry Annual Review*; April 2001.

[22] J.J. Labrosse, *MicroC/OS-II, The Real-time Kernel*, R & D Books, Oct. 1998.

[23] Analog Devices, http://www.analog.com/

[24] Wind River VSPWorks, technical brief, http://www.windriver.com/pdf/vspworks_tb.pdf

[25] http://web-cat.cs.vt.edu/PEBB/

[26] A.L. Davis and R.M. Keller, "Dataflow program graphs", *IEEE Computer*, vol. 15, no. 2, Feb. 1982, pp. 26-41.

[27] D.E. Culler, "Dataflow Architectures", *Annual Review of Computer Science*, vol. 1, Annual Reviews Inc., Palo Alto, CA, 1986.

[28] D. Garlan and M. Shaw, "An introduction to software architectures", in *Advances in Software Engineering and Knowledge Engineering*, vol. 1, World Scientific Publishing Co., New Jersey, 1993.

[29] Milan Milenkovic, "Operating Systems: Concepts and Design", McGraw-Hill Book Company, 1987.

[30] Jinghong Guo, Stephen H. Edwards, and Dushan Boroyevich, "Software Structure of the PEBB-based Plug and Play Power Electronics Systems", *16th IEE Applied Power Electronics Conference and Exposition*, Anaheim, CA, 2000.

[31] Jinghong Guo, Stephen H. Edwards, and Dushan Boroyevich, "Implementing dataflow-based control software for power electronics", In *Proceedings of the IEEE 9th Workshop on Computers in Power Electronics (COMPEL)*, 2002.

[32] Kuljeet Singh, "Design and Evaluation of an embedded real-time microkernel", M.S Thesis, Department of Computer Science, Virginia Tech, Oct. 2002, available online at: http://scholar.lib.vt.edu/theses/available/etd-11222002-121349/

[33] http://www.cuc.ucc.ie/research/overview/

Appendix - Code Listing

```
/*=========================================================================*\
| typedefs.h
|-------------------------------------------------------------------------
| Contains the type definitions used throughout the system
\*=========================================================================*
/

#ifndef TYPEDEFS_H
#define TYPEDEFS_H 1

#include <signal.h>
#include <ctype.h>
#include <stdlib.h>
#include <stdio.h>

#define NULL 0

typedef enum
{
  wait_for_fire =0   /* Process blocked on wait_to_fire operation  */
  ,  ready
   ,  blocked        /* Process blocked on read or write operation */
   ,  timed_wait     /* Process delayed without firing rule     */
     ,  timed_wait_for_fire  /* Process delayed with firing rule        */
   ,  dead
} ProcessState;

typedef enum
{
      no_actions = 0 /* Both timed wait and event queue empty */
   ,  future_actions /* timed wait actions pending          */
   ,  current_actions /* event queue actions pending        */
} Action_State;


typedef unsigned int  Time;
typedef unsigned int  Stack_Size;
typedef int        Inports_Num;
typedef int        Outports_Num;
typedef int        Cpcity;
typedef void*      ECO_Config;



/*-------------------------------------------------------------------------*\
 |  Thread Context
 *-------------------------------------------------------------------------*
 |  This structure is used to save the context of a thread during context
 | switching.
 |  The registers stored (in sequence) are:
 |  Data registers:
 |  MODE1
```

```
 |  R1,R2,R3,R5,  R6,R7,R9,R10,  R11,R13,R14,R15
 |  DAG:
 |  BO,IO,MO,LO,  B1,I1,M1,L1,  B2,I2,M2,L2,  B3,I3,M3,L3
 |  B5,I5,L5, B6,I6,L6,  B7,I7,L7
 |  SPECIAL REG:
 |  PCSTKP, LADDR
 |  INSTRUCTION POINTER:
 |  I12
\*-------------------------------------------------------------------------
*/
typedef struct
{
   int mode1;
   int r_reg[12];
   int dag1_reg[25];
   int spec_reg[2];
   int ip_reg;
} Context;

#endif
```

```
/*=============================================================================*\
| 160_cpp_hdr.asm
|-----------------------------------------------------------------------------
| This file implements the interrupt handler for external interrupts in
| Analog Devices SHARC assembly.
\*=============================================================================*/


#define __ADSP21160__
#define __2116x__
#undef  __ADSP21020__

#include "sig_glob.h"

#ifdef _ADI_THREADS
.EXTERN  __alloc_mutex__Fv;
.EXTERN  __dealloc_mutex__Fv;
#endif

.EXTERN  _atexit;
.EXTERN  ___process_needed_destructions__Fv;

#define  INT(irp)  \
 BIT CLR MODE1 0x1000;        /*Disable interrupts         */ \
 JUMP ___z3_int_determiner(DB); /*jmp to finish setting up   */ \
 DM(-1,I7)=R0;                /*Save r0 (scratch dreg)     */ \
 R0=SIGMASK(SIG_##irp);         /*Base of int table        */


#define RESERVED_INTERRUPT NOP;NOP;NOP;NOP

.segment/dm seg_dmda;
.var astat_store;
.var i5_store;
.var m4_store;
.var r5_store;
.var r6_store;
.var r7_store;
.endseg;


.GLOBAL      ___lib_prog_term;     /* Termination address         */
.GLOBAL      __done_execution;
.EXTERN      ___lib_setup_c;
.extern                __ctors;        /* CPP: this id is defined in ldf file
                                          which points to start of seg_ctdm
section */
.extern                __ctorsize;


.SEGMENT/PM       seg_rth;         /* Runtime header segment      */

      RESERVED_INTERRUPT;

___lib_RSTI:    NOP;          /* Not really executed         */
```

```
      JUMP ___lib_start;
      NOP;
      NOP;

___lib_IICDI:           INT(IICDI) ;    /* Access to illegal IOP space */

___lib_SOVFI:   INT(SOVF);       /* status/loop/PC stack overflow  */
___lib_TMZHI:   INT(TMZ0);       /* high priority timer         */
___lib_VIRPTI:    INT(VIRPTI);     /* external interrupts         */
___lib_IRQ2I:   INT(IRQ2);
___lib_IRQ1I:   INT(IRQ1);
___lib_IRQ0I:
      BIT CLR MODE1 0x1000;     /*Disable interrupts  */
       JUMP (pc,rest_of_code) (DB);   /*jmp to finish setting up*/
       !DM(I7,M7)=I15;         /*Save I15 (scratch reg)*/
       BIT SET MODE2 0x80000; /*freeze cache*/

       /* Saving the necessary registers */

rest_of_code:
        dm(r5_store)=r5;
      dm(r6_store)=r6;
      dm(r7_store)=r7;
      dm(astat_store)=ASTAT;
      dm(i5_store)=i5;
      dm(m4_store)=m4;

      /* ISR Body */

      R5 = dm(_event_code);/* Assuming that event code is already written in this
variable */
      I5 = _ev_queue_rear_inx;   /* Point to rear */
      R6 = dm(0,I5);   /* R6 contains rear of the queue */
      R6 = R6 + 1;     /* Advancing rear      */
      R7 = _ev_queue_max_len;
      COMP(R6,R7);
      IF NE JUMP(PC,2);
      R6 = 0;        /* Rotating rear   */
      M4 = R6;
      I5 = R4;
      dm(M4,I5) = R5;     /*Writing event code    */
      I5 = _ev_queue_rear_inx;
      dm(0,I5)=R6;     /* Recording rear*/
      R5=0x2;
      dm(_actions_pending) = R5; /* setting actions pending */

      /*Restoring the registers*/
      /*JUMP (PC,3) (DB,CI);*/  /* To allow nested interrupts of same kind */


      i5=dm(i5_store);
      m4=dm(m4_store);
      ASTAT=dm(astat_store);
      r5=dm(r5_store);
      r6=dm(r6_store);
      r7=dm(r7_store);
```

```
      /******finished********/

            RTI(DB);               /* Return from interrupt */
            BIT SET MODE1 0x1000; /* Re-enable Int */
      BIT CLR MODE2 0x00080000;/* Re-enable cache*/
      RESERVED_INTERRUPT;


___lib_SPR0I:   INT(SPR0I);      /* serial port DMA channel interrupts  */
___lib_SPR1I:   INT(SPR1I);
___lib_SPT0I:   INT(SPT0I);
___lib_SPT1I:   INT(SPT1I);
___lib_LP0I:            INT(LP0I);      /* link port DMA channel 4 */
___lib_LP1I:            INT(LP1I);      /* link port DMA channel 5 */
___lib_LP2I:            INT(LP2I);      /* link port DMA channel interrupts */
___lib_LP3I:            INT(LP3I);
___lib_LP4I:            INT(LP4I);      /* link port DMA channel 8 */
___lib_LP5I:            INT(LP5I);      /* link port DMA channel 9 */
___lib_EP0I:    INT(EP0I);       /* ext port DMA channel interrupts  */
___lib_EP1I:    INT(EP1I);
___lib_EP2I:    INT(EP2I);
___lib_EP3I:    INT(EP3I);
___lib_LSRQ:    INT(LSRQ);       /* link service request        */
___lib_CB7I:    INT(CB7);     /* circular buffer #7 overflow */
___lib_CB15I:   INT(CB15);       /* circular buffer #15 overflow*/
___lib_TMZLI:   INT(TMZ);     /* low priority timer          */
___lib_FIXI:    INT(FIX);     /* fixed point overflow        */
___lib_FLTOI:   INT(FLTO);       /* floating point overflow     */
___lib_FLTUI:   INT(FLTU);       /* floating point underflow    */
___lib_FLTII:   INT(FLTI);       /* floating point invalid      */
___lib_SFT0I:   INT(USR0);       /* user interrupts 0..3        */
___lib_SFT1I:   INT(USR1);
___lib_SFT2I:   INT(USR2);
___lib_SFT3I:   INT(USR3);
      RESERVED_INTERRUPT;

___z3_int_determiner:
      MODIFY(I7,-3);
      DM(I7,-2)=R1;
      R1=I2;
      DM(I7,-2)=R1;           /* Save I2 (scratch reg)       */
      I2=R0;
      DM(0,I7)=I13;           /* Save I13 (scratch reg)      */
      MODIFY(I7,-2);
      I13=DM(5,I2);           /* get disp to jump to         */
      JUMP (M13, I13) (DB);   /* Jump to dispatcher          */
      BIT SET MODE2 0x80000;    /* Freeze cache              */
      I13=DM(2,I2);           /* rd handler addr (base+2)    */

/* Note:  It's okay to use PM in getting the above values b'cse z3 has a */
/* linear memory. Therefore dm and pm are the same and we can use either.*/

___lib_start:
      CALL ___lib_setup_c;      /* Setup C runtime model       */

/* _lib_call_ctors is the added code to support the constructor calls
```

```
before the main call. if we are using __ctorsize, then we have to change
this code. */
#ifdef _ADI_THREADS
/* call alloc_mutex */
                        r2 = i6;
                        i6 = i7;
                        jump  __alloc_mutex__Fv (DB);
                        dm(i7,m7)=r2;  /* store frame pointer */
                        dm(i7,m7)=pc;  /* store PC */
#endif
      i0 = __ctors;
_lib_call_ctors:
      m12 = 0x0;
      m4 = 0x1;
      r0 = dm(i0,m4);    /* get the address of constructor function.*/
      r0 = pass r0;
      if eq jump _lib_call_ctors_exit; /* check if null */
      i13=r0;
/* next 5 lines are for call of the constructor */
      r2 = i6;
      i6 = i7;
      jump(m12,i13) (db);
      dm(i7,m7)=r2;
      dm(i7,m7)=pc;
      jump _lib_call_ctors;
_lib_call_ctors_exit:
#ifdef _ADI_THREADS
         r4=__dealloc_mutex__Fv ;
         r2 = i6;
         i6 = i7;
         JUMP  _atexit (DB);
           dm(i7,m7)=r2;  /* store frame pointer */
           dm(i7,m7)=pc;  /* store PC */
#endif

         /* perform static c++ destructors at exit of main */
         r4=___process_needed_destructions__Fv;
         r2 = i6;
         i6 = i7;
         JUMP  _atexit (DB);
           dm(i7,m7)=r2;  /* store frame pointer */
           dm(i7,m7)=pc;  /* store PC */

#ifdef MAIN_RTS
      CJUMP _main (DB);         /* Begin C program          */
      DM(I7,M7)=R2;
      DM(I7,M7)=PC;
#else
      JUMP _main;             /* Begin C program          */
#endif

/* Setting the __done_execution flag indicates that this processor is  */
/* finished executing, for the benefit of anyone who may be watching.  */

___lib_prog_term: PM(__done_execution)=PC;
      IDLE;
      JUMP ___lib_prog_term;    /* Stay put                 */
```

```
.VAR __done_execution = 0;

.ENDSEG;

#ifdef _ADI_SWFA
.SEGMENT/PM       seg_pmda;
.GLOBAL __21160_anomaly_write_location;
.VAR __21160_anomaly_write_location;   // A memory location we can write to
                                       // to flush the FIFO.
#endif
.ENDSEG;
```

```
/*=========================================================================*\
| Byte_Data_Channel.h
|-------------------------------------------------------------------------
| Header file for Byte Data Channel
\*=========================================================================*/

#ifndef BYTE_DATA_CHANNEL_H
#define BYTE_DATA_CHANNEL_H 1

#include "Data_Channel.h"


class Byte_Data_Channel : public Data_Channel
{
 public:
      Byte_Data_Channel();
      Byte_Data_Channel(enum Overflow_Style , int , bool );
      ~Byte_Data_Channel();

/*-------------------------------------------------------------------------*
 | Method to write given array of bytes of specified length to the
 | data channel - calls base class method
\*-------------------------------------------------------------------------*/
      inline void Write(char* in_buf, int length)
      {
        Write_bytes(in_buf, length);
      }


/*-------------------------------------------------------------------------*
 | Method to read specified number of bytes from the data channel - calls
 | base class method
\*-------------------------------------------------------------------------*/
      inline void Read(char* out_buf, int length)
      {
        Read_bytes(out_buf, length);
      }

};

#endif
```

```
/*=========================================================================*\
| Byte_Data_Channel_I.cpp
|-------------------------------------------------------------------------
| Implementation file for Byte Data Channel.
\*=========================================================================*/

#include "Byte_Data_Channel.h"

/*-------------------------------------------------------------------------*\
|  Constructors and destructor for Byte_Data_Channel
\*-------------------------------------------------------------------------*/

Byte_Data_Channel :: Byte_Data_Channel()
    : Data_Channel(10, 1, OS_Overwrite_Oldest, false)
{
 byte_dc = true;

}

Byte_Data_Channel :: Byte_Data_Channel(enum Overflow_Style os, Cpcity cap, bool
interrupt)
 : Data_Channel(10, cap, os, interrupt)
{
 byte_dc = true;
}

Byte_Data_Channel :: ~Byte_Data_Channel()
{
}
```

```
/*==========================================================================*\
| DARKpp.h
|--------------------------------------------------------------------------
| Kernel header file
\*==========================================================================*/

#ifndef DARKPP_H
#define DARKPP_H 1

#include "typedefs.h"

class ECO;
class Data_Channel;


/*--------------------------------------------------------------------------*
 | Kernel class - singleton
\*--------------------------------------------------------------------------*/
class DARKpp
{
 friend ECO;

 public:

      DARKpp(int, int );

      ~DARKpp();

/*--------------------------------------------------------------------------*\
 | Method returns reference to a static kernel object - used every time
 | a kernel method has to be called from the ECO thread.
\*--------------------------------------------------------------------------*/

      inline static DARKpp& getInstance(int num_processes, int num_data_channels)
      {
        static DARKpp the_kernel(num_processes, num_data_channels);
        return the_kernel;
      }

      void register_OS(ECO* );
      void register_OS(Data_Channel* );

      int  NumProcesses();

      void execute();


/*--------------------------------------------------------------------------*\
 | Method returns reference to the environment information for the kernel
 | thread - used to switch context from process thread to kernel
\*--------------------------------------------------------------------------*/
      inline Context* OS_Env()
      {
        return(&OSenv);
      }
```

```
 private:

      void make_process_ready_and_run(ECO* );

      int    num_ECOs; // number of ECOs
      int    num_DCs;  // number of Data Channels

      bool startup_completed;
      bool first_process_turn;
      bool process_changed;

      int    num_ECO;
      int    num_DC;

      ContextOSenv;


/*-------------------------------------------------------------------------*
 | Arrays for the kernel to keep track of the processes and data channels
 | in the application - processes and data channels register themselves with
 | the kernel as soon as they are created.
\*-------------------------------------------------------------------------*/
      ECO**       OS_All_Processes;
      Data_Channel**  OS_All_Data_Channels;
};

void add_to_waiting_queue(ECO* );

#endif
```

```
/*=========================================================================*\
| DARKpp_cfg.h
|-------------------------------------------------------------------------
| Configuration file for DARK++.
| Behaviour can be adjusted according to the real time needs of the
| applications.
\*=========================================================================*/

#ifndef DARKPP_CFG_H
#define DARKPP_CFG_H 1

/*----------------- Kernel Options available to the user -----------------*/

/* Preemptive and multithreaded DARK++ with firing rules & dynamic scheduling
*/
#define    PREEMPTIVE_MTHREADED    1

/* Non-Preemptive and multithreaded DARK++ with firing rules & dynamic scheduling
*/
#define    NONPREEMPTIVE_MTHREADED   0

/* Dynamically scheduled, single threaded DARK++ with firing rules         */
#define    SINGLETHREAD_DYNSCHD    0

/* Statically scheduled, single threaded DARK++ with no firing rules       */
#define    SINGLETHREAD_STATSCHD   0

/*-------------------------------------------------------------------------*/


/* Adjusting the general properties of the OS according to the user's selection */
#if   (PREEMPTIVE_MTHREADED)
 #define PREEMPTIVE 1
 #define MTHREADED  1
 #define DYNSCHD    1
#elif (NONPREEMPTIVE_MTHREADED)
 #define PREEMPTIVE 0
 #define MTHREADED  1
 #define DYNSCHD    1
#elif (SINGLETHREAD_DYNSCHD)
 #define PREEMPTIVE 0
 #define MTHREADED  0
 #define DYNSCHD    1
#elif (SINGLETHREAD_STATSCHD)
 #define PREEMPTIVE 0
 #define MTHREADED  0
 #define DYNSCHD    0
#endif


#endif
```

```
/*=========================================================================*\
| DARKpp_I.cpp
|-------------------------------------------------------------------------
| This file implements the most important functions of the kernel. It is the
| core file of thee kernel.
\*=========================================================================*/

#include "Data_Channel.h"
#include "typedefs.h"
#include "Event_Handler.h"
#include "Event_Handlers.h"
#include "Event_Queue.h"
#include "DARKpp_cfg.h"
#include "Waiting_Queue.h"
#include <iostream.h>


extern "C" void shallow_longjmp(Context* , int );
extern "C" void longjmp_all(Context* , int );
extern "C" void convert_shallow_setjmp_to_all(Context* );
extern "C" int setjmp_all(Context* );
extern "C" int shallow_setjmp(Context* );
extern "C" int setjmp_initialize(Context* );
static void start_ECO(ECO* );
extern Event_Queue the_event_queue;
Waiting_Queue the_waiting_queue = Waiting_Queue::getInstance();

/*-------------------------------------------------------------------------*\
| To handle timed events & interrupts.
| Made volatile because it can altered by scheduler/process threads
\*-------------------------------------------------------------------------*/
volatile Action_State actions_pending = no_actions;

volatile int current_time; // will be set by ISR


ECO*  Current = NULL;  // Indicates currently executing ECO.


DARKpp :: DARKpp(int num_processes, int num_data_channels)
{
 num_ECOs = num_processes;
 num_DCs = num_data_channels;
 OS_All_Processes = new ECO*[num_ECOs];
 OS_All_Data_Channels = new Data_Channel*[num_data_channels];
 startup_completed = false;
 first_process_turn = false;
 num_ECO = 0;
 num_DC =0 ;
}


DARKpp :: ~DARKpp()
{
 delete[] OS_All_Processes;
 delete[] OS_All_Data_Channels;
}
```

```
int DARKpp :: NumProcesses()
{
 return num_ECOs;
}


/*-------------------------------------------------------------------------*\
|  Scheduler and dispatcher for all four versions of the kernel -
| full-featured, non-preemptive, single-threaded dynamically scheduled and
| single-threaded statically scheduled. Uses the selected preprocessor
| directives from DARKpp_cfg.h to take appropriate action.
\*-------------------------------------------------------------------------*/
void DARKpp :: execute()
{
 int i=0, index = 0;

 process_changed = true; // Initialized to true to account for first process


#if (MTHREADED)
 for ( int j = 0; j < num_ECOs; j++ )
 {
   make_process_ready_and_run( OS_All_Processes[j] );
   OS_All_Processes[j]->SetProcess_State(wait_for_fire);
 }
#endif

 (*(event_handlers[0]))();


 //Start running the processes && INFINITE OS LOOP

 while (true)
 {
   if (i == num_ECOs)
   {
     i = 0;
     (*(event_handlers[0]))();
   }

   /* handle interrupts and other actions */
   if (actions_pending != no_actions)
   {
     if (the_waiting_queue.Front() != the_waiting_queue.Rear())
     {
       //handle_time_arrows();
     }
     if (the_waiting_queue.Front() != the_waiting_queue.Rear())  // Not empty
     {
       actions_pending = future_actions;
     }
     else
     {
       actions_pending = no_actions;
     }
     if (the_event_queue.Front() != the_event_queue.Rear())
```

```
      {
        //handle_all_events();
      }
    }

    i++;

/*-------------------- For the Multi-threaded DARK++ ----------------------*/
#if (MTHREADED)
      if ( the_ready_queue.get_num_entries() > 0 )
      {
        if (Current == NULL)
        {
          Current = the_ready_queue.remove_topofheap();

/* to decide whether to do shallow or full context save */
          process_changed = true;

        }
        else if
        ((Current->Current_Priority() <= the_ready_queue.next_available_ECO()-
>Current_Priority())
             || (Current->Process_State() != ready)
             && !process_changed )
        {

/* change current process && NOT process_changed added to leave the first
process. This case will arise only in case of an interrupt or after system call */

          if (Current->Process_State() == ready)
          {

/* Save full context for the current process from the alternate register set
   In other words convert shallow_setjmp buffer into a setjmp_all buffer
   NEED TO CALL shallow_setjmp IN EACH ISR  */

            convert_shallow_setjmp_to_all(Current->ECO_Env());
            the_ready_queue.insert(Current);
          }
          Current = the_ready_queue.remove_topofheap();
          process_changed = true;
        }

        if ( actions_pending != current_actions )
        {
          if ( process_changed )
          {
            process_changed = false;
            if ( shallow_setjmp(&OSenv) == 0 )
            {
              longjmp_all(Current->ECO_Env(),1);

            }

          }
          else
          {
```

```
            if (Current->Process_State() == ready) //to take care of "not-ready"
                                                   current processes
            {
              if ( shallow_setjmp(&OSenv) == 0 )
              {
                shallow_longjmp(Current->ECO_Env(),1);
              }
            }
           }
        } //if (!actions_pending)
      }
      else if (Current->Process_State() == ready) //to take care of "ready" current
                                                    processes when queue_size=0
      {
        if ( shallow_setjmp(&OSenv) == 0 )
        {
          shallow_longjmp(Current->ECO_Env(),1);
        }
      }
#endif

/*-------------- For Single Threaded and dynamically scheduled DARK++ --------*/
#if (!MTHREADED && DYNSCHD)
      if ( the_ready_queue.get_num_entries() > 0 )
      {
        Current = the_ready_queue.remove_topofheap();
        start_ECO(Current);
        Current->swap(wait_for_fire);
      }
#endif

/*-------------- For Single Threaded and statically scheduled DARK++ -------*/
#if (!MTHREADED && !DYNSCHD)
      if (index == num_ECOs)
      {
        index = 0;
      }

      Current = OS_All_Processes[index++];

      if (Current != NULL)
      {
        start_ECO(Current);
      }
#endif

 }//while(true)
}


void DARKpp :: register_OS(ECO* p)
{
 OS_All_Processes[num_ECO++] = p;
}


void DARKpp :: register_OS(Data_Channel* dc)
```

```
{
 OS_All_Data_Channels[num_DC++] = dc;
}



/*-------------------------------------------------------------------------*\
|Method to initialize the stack pointers and start the running the process for the
|   first time
\*-------------------------------------------------------------------------*/
void DARKpp :: make_process_ready_and_run(ECO* eco)
{
 if (eco == NULL)   //In case of ISRs
 {
   return;
 }

 unsigned int *ptrfront_stk = eco->Stack();
 unsigned int *ptrend_stk = eco->Stack() + eco->StackSize() - 1;
 Stack_Size local_stacksize = eco->StackSize();
 Context* local_ecocontext = eco->ECO_Env();


 if ( setjmp_all(&OSenv) != 0 )
 {
   return;
 }


 asm( "r1 = %0;" : : "d" (local_ecocontext) : "r1" );
 asm( "r5 = %0;" : : "d" (local_stacksize) : "r5");
 asm( "r9 = %0;" : : "d" (ptrfront_stk) : "r9");
 asm( "r10 = %0;" : : "d" (ptrend_stk) : "r10");



 // Initializing the stack pointers
 asm("b6 = r9;");
 asm("b7 = r9;");
 asm("l6 = r5;");
 asm("l7 = r1;");
 asm("I6 = r10;");
 asm("I7 = r10;");


 if ( setjmp_initialize(local_ecocontext) == 0 )
 {
   longjmp_all(&OSenv,1);
 }

 // Execute the ECO Implementation method
 start_ECO(Current);
 Current->swap(dead);
}
```

```
/*-------------------------------------------------------------------------*\
|Function to start running process by calling its implementation method -
|called by the single-threaded versions of the scheduler.
\*-------------------------------------------------------------------------*/
inline static void start_ECO(ECO* eco)
{
 eco->Implementation();
}

/*-------------------------------------------------------------------------*\
|Function to insert an ECO into ready queue - called by ECO - function required
|   to overcome the problem of mutual header file inclusion - Waiting_Queue.h and
|ECO.h - these classes call each others' methods and the Waiting Queue object is
|required to be global
\*-------------------------------------------------------------------------*/
inline void add_to_waiting_queue(ECO* eco)
{
 the_waiting_queue.insert(eco);
}
```

```
/*=========================================================================*\
| Data_Channel.h
|-------------------------------------------------------------------------
| Header file for base class - Data_Channel
\*=========================================================================*/

#ifndef DATA_CHANNEL_H
#define DATA_CHANNEL_H 1

#include "ECO.h"
#include "DARKpp.h"
#include "Ready_Queue.h"
#include "typedefs.h"
#include "DARKpp_cfg.h"
#include <iostream.h>
#include <string.h>


extern ECO* Current; //Currently executing process

extern volatile Action_State actions_pending; //current state

extern "C" int  shallow_setjmp(Context* );
extern "C" void shallow_longjmp(Context* , int );


extern DARKpp the_kernel; // kernel object

#if (DYNSCHD)
extern ready_queue the_ready_queue;  // ready queue object
#endif

/*-------------------------------------------------------------------------*\
| Macro returns true if data channel is empty - used to check if
| requesting process has to be blocked on read
\*-------------------------------------------------------------------------*/
#define empty(data_channel) ( !( data_channel->Entries() ) )


/*-------------------------------------------------------------------------*\
| Macro returns true if data channel is full - used to check if
| data channel is blocked on write
\*-------------------------------------------------------------------------*/
#define full(data_channel)  (data_channel->Capacity() == data_channel->Entries())


/*-------------------------------------------------------------------------*\
| For non-interrupt-driven data channels, checks if there is an equal or
| higher priority ready process at the end of the Read operation and switches
| to kernel is there is
\*-------------------------------------------------------------------------*/
#if (PREEMPTIVE)
     #define READ_GOTO_OS(interrupt)\
     {\
        if(interrupt == false && (actions_pending != no_actions || \
            ((the_ready_queue.get_num_entries() > 0  && \
```

```
              (Current->Current_Priority() >= the_ready_queue.next_available_ECO()-
                        >Current_Priority()))))\
    {\
            if ( shallow_setjmp( Current->ECO_Env() ) == 0 ) \
                shallow_longjmp(the_kernel.OS_Env(),1);    \
        }\
 }
#else
     #define READ_GOTO_OS(interrupt)
#endif


/*-------------------------------------------------------------------------*\
| Macro sets an empty data channel Blocked and waits until it has data
\*-------------------------------------------------------------------------*/
#if (DYNSCHD)
    #define BLOCK_ON_READ(data_channel) \
    { \
      while (empty(data_channel))\
    {   \
      data_channel->setBlocked(); \
      Current->swap(blocked); \
    }\
    }
#else
    #define BLOCK_ON_READ(data_channel)
#endif


/*-------------------------------------------------------------------------*\
| For non-interrupt-driven data channels, checks if there is an equal or
| higher priority ready process at the end of the Write operation and switches
| to kernel is there is
\*-------------------------------------------------------------------------*/
#if (PREEMPTIVE)
 #define WRITE_GOTO_OS(interrupt)  \
 {                        \
    if (interrupt_driven == false && ( actions_pending != no_actions ||   \
      ( (the_ready_queue.get_num_entries() > 0)  && ( Current != NULL ) ) ) )    \
    {                     \
      if(Current->Current_Priority() >= the_ready_queue.next_available_ECO()-
>Current_Priority())\
      {                   \
        if ( shallow_setjmp( Current->ECO_Env() ) == 0 )\
        {                                \
             shallow_longjmp(the_kernel.OS_Env(),1);    \
        }\
      }\
    }                                 \
 }
#else
    #define WRITE_GOTO_OS(interrupt)
#endif
```

```
/*-------------------------------------------------------------------------*\
| Macro for overflow handling
\*-------------------------------------------------------------------------*/
#if (DYNSCHD)
    #define BLOCK_ON_WRITE( data_channel, length )\
    { \
    while(full(data_channel))\
    { \
            switch (data_channel->overflow_style) \
            { \
                case OS_Block  :  /*block*/ \
                                  data_channel->setBlocked(); \
                                  Current->swap( blocked ); \
                                  break; \
                case OS_Overwrite_Newest: /*re-adjust rear*/ \
                                       data_channel->rear -= length; \
                                       if (data_channel->rear < 0) \
                                       { \
                                          data_channel->rear += size; \
                                       } \
                                       break; \
               case OS_Overwrite_Oldest: \
                                       data_channel->front += length; \
                                       if(data_channel->front == size) \
                                       { \
                                          data_channel->front = 0; \
                                       } \
                                       break; \
         } \
      }\
    }
#else
    #define BLOCK_ON_WRITE(data_channel, length)
#endif
```

```
class Data_Channel
{
 protected:

/*-------------------------------------------------------------------------*\
| Indicates action to be taken if data channel is full
| Has been defined here because the second constructor declared in public
| (see below) requires this definition to be here rather than later along
| with the declaration of all protected data members
\*-------------------------------------------------------------------------*/
    enum Overflow_Style
      { OS_Block = 0, OS_Overwrite_Newest, OS_Overwrite_Oldest } ;



 public:

/*-------------------------------------------------------------------------*\
| Data_Channel constructor calls protected ECO methods -
| Register_As_Source() and Register_As_Sink(), therefore a friend of ECO
\*-------------------------------------------------------------------------*/
      friend class ECO;


      Data_Channel(int, int ,enum Overflow_Style , bool );
      ~Data_Channel();

/*-------------------------------------------------------------------------*\
| Method returns capacity in terms of number of data elements
\*-------------------------------------------------------------------------*/
      inline int  Capacity()
      {
        return capacity;
      }


/*-------------------------------------------------------------------------*\
| Method returns number of data elements present in the data channel
\*-------------------------------------------------------------------------*/
      inline int   Entries()
      {
        return num_entries;
      }


/*-------------------------------------------------------------------------*\
| Method returns available capacity in bytes
\*-------------------------------------------------------------------------*/
      inline int  Available_Capacity()
      {
        return (size - Bytes());
      }
```

```
/*-------------------------------------------------------------------------*\
| Method removes all entries in the data channel
\*-------------------------------------------------------------------------*/
      inline void Flush(int num_entries)
      {
        if (byte_elems == 0)
        {
          front = 0;
          rear = 0;
        }
        else
        {
          // Deleting first "num_entries" items

          int i;
          int num_bytes_to_delete =
            num_entries * element_size;

          for (i=0; i<num_bytes_to_delete; i++)
          {
            front = (front + 1) % (size +1);
          }
        }
      }


/*-------------------------------------------------------------------------*\
| Method returns true if data_channel has an overflow style of "Blocked"
| and is full
\*-------------------------------------------------------------------------*/
      inline bool Blocked()
      {
        return (block);
      }


/*-------------------------------------------------------------------------*\
| Method returns number of bytes present in the data channel
\*-------------------------------------------------------------------------*/
      inline int  Bytes()
      {
        if (rear >= front)
          return (rear - front);
        else
          return (size + rear - front);
      }

 protected:
 inline void setBlocked()
 {
   block = true;
 }

 inline void resetBlocked()
 {
   block = false;
 }
```

```
/*-------------------------------------------------------------------------*\
| Method resets the bit corresponding to this port in the firing mask of the
| source ECO
\*-------------------------------------------------------------------------*/
#if (DYNSCHD)

 inline void setmask_on_read()
 {

     /* Mask needs to be reset only if queue becomes empty */
     if ( front == rear )
     {
      sink_ECO->SetIn_Ports_Ready(sink_ECO->In_Ports_Ready()
        & ~(1 << sinkport_num));
   }

     //Freeing the source process if it is on a write block
     if (source_ECO != NULL && Blocked() && source_ECO->Blocked())
     {
      resetBlocked();
      source_ECO->SetProcess_State(ready);
      the_ready_queue.insert(source_ECO);
     }
 }
#else

 #define setmask_on_read()

#endif



/*-------------------------------------------------------------------------*\
| Method sets the bit corresponding to this port in the firing mask of the
| sink ECO
\*-------------------------------------------------------------------------*/
#if (DYNSCHD)

 inline void setmask_on_write()
 {
     ECO*    snk_ECO = sink_ECO;

     FiringRule temp_rule   = snk_ECO->FIRE_Rule();

   snk_ECO->SetIn_Ports_Ready(sink_ECO->In_Ports_Ready()
    | (1 << sinkport_num));

   switch ( snk_ECO->Process_State() )
   {
     case dead:
     case timed_wait:
          return;

     case blocked:
          if ( Blocked() )
          {
            resetBlocked();
```

```
            snk_ECO->SetProcess_State(ready);
            the_ready_queue.insert(snk_ECO);
          }
          return;

      case ready:
        while (!temp_rule->Last())
        {
            if((snk_ECO->In_Ports_Ready() & temp_rule->Firing_Mask()) == temp_rule-
               >Firing_Mask())
             {
             snk_ECO->SetWakeup_Call(temp_rule->Firing_Mask());
             the_ready_queue.change_value(snk_ECO, temp_rule->Priority());
             return;
          }
          temp_rule++;
        } //end while

        return;

      default:
        while (!temp_rule->Last())
        {
          if
          ((snk_ECO->In_Ports_Ready() & temp_rule->Firing_Mask())
            == temp_rule->Firing_Mask())
          {
            snk_ECO->SetWakeup_Call(temp_rule->Firing_Mask());
            snk_ECO->SetProcess_State(ready);
            the_ready_queue.insert(snk_ECO);
            return;
          }
          temp_rule++;
        } // end while
        return;
     }
 }
#else

 #define setmask_on_write()

#endif


/*-------------------------------------------------------------------------*\
| Method reads specified number of bytes from the data channel into the
| array passed.
\*-------------------------------------------------------------------------*/
 inline void Read_bytes(char* data, int length)
 {
      unsigned int i; //index

      if(length <= size) // ASSERT
      {
        BLOCK_ON_READ(this);

        for (i=0; i<length; i++)
```

```
        {
          // add 1 to account for 1 byte of the buffer array
          front = (front + 1) ;

          if (front == size)
            front = 0;

          *(data+i) = *(buffer + front);
        }

        num_entries = Bytes() / element_size;

        setmask_on_read();

        READ_GOTO_OS(interrupt_driven);
      }
      else
        cout << "ASSERT failed." << endl << flush;
 }



/*-------------------------------------------------------------------------*\
| Method writes to the data channel, given data of specified length
| (number of bytes).
\*-------------------------------------------------------------------------*/
 inline void Write_bytes(char* data, int length)
 {
        unsigned int i; //index
        int temp_rear;
        unsigned int rem_cap_buffer;

        if(length <= size)
        {
          BLOCK_ON_WRITE(this, length);

          //writing the bytes
          for (i=0; i<length; i++)
          {
               rear =  (rear + 1);// % (size + 1);
               if (rear == size)
                  rear = 0;

               *(buffer + rear) = *(data++);
          }

          num_entries = Bytes() / element_size;

          setmask_on_write();

          WRITE_GOTO_OS(interrupt_driven);
        }

 } // end Write_bytes


 void Register_OS();
```

```
 volatile char* buffer;     // buffer to hold the data - circular queue
 int   element_size; // number of bytes required to store the particular data type


/*-------------------------------------------------------------------------*\
| front and rear used for queue arithmetic.
| Volatile because scheduler can potentially call read/write (on
| interrupt-driven data channels).
\*-------------------------------------------------------------------------*/
 volatile int  front;
 volatile int  rear;


 int  size;       // allocated size of the queue
 int  byte_elems; // number of bytes in the data channel
 bool byte_dc;   // indicates whether a byte data channel



/*-------------------------------------------------------------------------*\
| Data member when set, indicates whether channel is blocked.
| Set/reset in setmask_on_read and setmask_on_write.
| Volatile since the read/write operations can be called from process/kernel
| threads.
\*-------------------------------------------------------------------------*/
 volatile bool block;


/*-------------------------------------------------------------------------*\
| Pointers to source and sink ECO objects
\*-------------------------------------------------------------------------*/
 ECO* source_ECO;
 ECO* sink_ECO;


/*-------------------------------------------------------------------------*\
| Index in the input port and output port arrays in the corresponding
| source and sink ECOs
\*-------------------------------------------------------------------------*/
 short int sourceport_num;
 short int sinkport_num;


 enum Overflow_Style overflow_style;

 bool interrupt_driven;

 int num_entries;
 int capacity;

};


#endif
```

```
/*=========================================================================*\
| Data_Channel_I.cpp
|-------------------------------------------------------------------------
| Implementation file for Data Channel.
\*=========================================================================*/

#include "Data_Channel.h"
#include <string.h>
#include <iostream.h>


/*-------------------------------------------------------------------------*\
|  subclasses call this constructor with the required size of buffer
|  (in bytes), appropriate element size for the particular data type and
|   required overfow style
\*-------------------------------------------------------------------------*/
Data_Channel :: Data_Channel(int sz, int es, enum Overflow_Style os, bool
interrupt)
{
 front = 0;
 rear = 0;

 num_entries = 0;

 capacity = sz / es;

 byte_dc = false;
 element_size = es;
 size = sz;
 buffer = new char[size];
 block = false;
 overflow_style = os;
 interrupt_driven = interrupt;

 Register_OS();
}



Data_Channel :: ~Data_Channel()
{
 delete[] buffer;
}


/*-------------------------------------------------------------------------*\
|   Method to register data channel with kernel (in OS_All_Data_Channels).
\*-------------------------------------------------------------------------*/
void Data_Channel :: Register_OS()
{
 the_kernel.register_OS(this);
}
```

```
/*=========================================================================*\
| ECO.h
|-------------------------------------------------------------------------
| Header file for base class - ECO
\*=========================================================================*/

#ifndef ECO_H
#define ECO_H 1

#include "DARKpp.h"
#include "typedefs.h"
#include "Priority_Firing_Mask.h"
#include "DARKpp_cfg.h"


extern "C" void shallow_longjmp(Context* , int );
extern "C" int setjmp_all(Context* );
extern void add_to_waiting_queue(ECO* );
extern DARKpp the_kernel;
extern volatile int current_time;
extern volatile Action_State actions_pending;
extern ECO* Current;

class Data_Channel;

class ECO
{
 public:
      ECO();

      ECO(Inports_Num ,Outports_Num ,FiringRule , Prty ,Stack_Size ,Data_Channel**
,Data_Channel** );

      ~ECO()
      {

      }

      virtual void  Implementation()=0;
      unsigned int* Stack();
      Stack_Size  StackSize();
      void   Register_OS();


/*-------------------------------------------------------------------------*\
 |  ECO Process Management Operation: This method waits for a firing rule
 |  to go true.  The write function is responsible for removing this block.
 *-------------------------------------------------------------------------*/
#if (MTHREADED)
      inline bool   Wait_To_Fire()
        {
            FiringRule temp_rule   = F_Rule;

          while (!temp_rule->Last())
            {
             if (in_ports_ready == temp_rule->Firing_Mask() )
```

```
               return true;

             temp_rule++;
            }
          swap(wait_for_fire);
            return true;
        }
#else
      #define Wait_To_Fire() false
#endif


/*-------------------------------------------------------------------------*\
|   Method that returns the current pririoty of the process.
\*-------------------------------------------------------------------------*/
      inline Prty   Current_Priority()
      {
        return ECOpriority;
      }


/*-------------------------------------------------------------------------*\
|   Method that sets the pririoty of the process.
\*-------------------------------------------------------------------------*/
      inline void   SetPriority(Prty new_priority)
        {
        ECOpriority = new_priority;
      }


/*-------------------------------------------------------------------------*\
|   Method that sets the inports_ready data member - called at the end of
|   every write operation to set the firing mask of the sink ECO of the port
|   written to.
\*-------------------------------------------------------------------------*/
      inline void   SetIn_Ports_Ready(FiringMask inports_ready)
      {
        in_ports_ready = inports_ready;
      }



/*-------------------------------------------------------------------------*\
|   Method that returns the in_ports_ready data member of the process.
\*-------------------------------------------------------------------------*/
      inline FiringMask  In_Ports_Ready()
      {
        return(in_ports_ready);
      }



/*-------------------------------------------------------------------------*\
|   Method that sets the process state as specified.
\*-------------------------------------------------------------------------*/
      inline void   SetProcess_State(ProcessState pstate)
      {
```

```
        process_state = pstate;
      }


/*-------------------------------------------------------------------------*\
|   Method that returns true if the process is blocked waiting for data.
\*-------------------------------------------------------------------------*/
      inline bool   Blocked()
      {
        if(process_state == blocked)
          return true;
        else
          return false;
      }


/*-------------------------------------------------------------------------*\
|   Method that sets the firing mask that will wake up the process.
|Called at the end of a write operation if the process is found to be ready.
\*-------------------------------------------------------------------------*/
      inline void   SetWakeup_Call(FiringMask fmask)
      {
        wakeup_call = fmask;
      }


/*-------------------------------------------------------------------------*\
|   Method that returns the firing mask that woke the process up.
\*-------------------------------------------------------------------------*/
      inline FiringMask  Wakeup_Call()
      {
        return(wakeup_call);
      }


/*-------------------------------------------------------------------------*\
|   Method that sets the time at which process is to wake up.
\*-------------------------------------------------------------------------*/
      inline void   SetWakeup_Time(int wkup_time)
      {
        wakeup_time = wkup_time;
      }


/*-------------------------------------------------------------------------*\
|   Method that returns the time at which the process is to wake up.
\*-------------------------------------------------------------------------*/
      inline int  Wakeup_Time()
      {
        return(wakeup_time);
      }



/*-------------------------------------------------------------------------*\
|   Method that returns the current state of the process.
```

```
\*-------------------------------------------------------------------------*/
      inline ProcessStateProcess_State()
      {
        return(process_state);
      }


      inline FiringRule  FIRE_Rule()
      {
        return(F_Rule);
      }



/*-----------------------------------------------------------------------------*\
|Method to swap the process with the process on the head of the
|   ready queue
\*-----------------------------------------------------------------------------*/
      inline void   swap(ProcessState next_state)
      {
        Current->SetProcess_State(next_state); // Can be other state also like
waiting

#if (MTHREADED)
           if (next_state == dead)
           {
             shallow_longjmp(the_kernel.OS_Env(),1);
           }
           if (setjmp_all(&(ECOenv)) == 0)
           {
             shallow_longjmp(the_kernel.OS_Env(),1);
           }
#endif

        return;
      }


/*-------------------------------------------------------------------------*\
 | Method returns reference to the environment information for the process
 | thread - used to switch context from kernel to the process thread
\*-------------------------------------------------------------------------*/
      inline Context* ECO_Env()
      {
        return(&ECOenv);
      }



/*-------------------------------------------------------------------------*\
|  ECO Process Management Operation: This method waits for a firing rule
|  to go true OR delays for a certain time . Two conditions can remove this
|   block:
|  + A firing rule goes off.
|  + The timed wait expires.
*-------------------------------------------------------------------------*/
       inline bool Timed_Wait_To_Fire(int delay_time)
```

```
       {
          wakeup_time = delay_time + current_time;
          actions_pending = future_actions;
          add_to_waiting_queue(this);
          swap(timed_wait_for_fire);
          return true;
       }


/*-------------------------------------------------------------------------*\
|  ECO Process Management Operation: This method delays the process for
|  the given time
*-------------------------------------------------------------------------*/
       bool Delay(int delay_time)
       {
          wakeup_time = delay_time + current_time;
          actions_pending = future_actions;
          add_to_waiting_queue(this);
          swap(timed_wait);
          return true;
       }



 protected:

/*-------------------------------------------------------------------------*\
| array of pointers to input and output data channels
*-------------------------------------------------------------------------*/
      Data_Channel** inports;
      Data_Channel** outports;


/*-------------------------------------------------------------------------*\
| number of inout and output ports (data channels)
*-------------------------------------------------------------------------*/
      Inports_Num   inports_num;
      Outports_Num  outports_num;


/*-------------------------------------------------------------------------*\
| ECO configuration info - void* - exact structure diff. for diff. ECOs
| and to be defined in the specific ECO
*-------------------------------------------------------------------------*/
      ECO_Config    config;



/*-------------------------------------------------------------------------*\
| Firing Rule for the ECO
*-------------------------------------------------------------------------*/
      FiringRule    F_Rule;
```

```
/*-------------------------------------------------------------------------*\
| registering as source and sink of appropriate output/input
| data channels, respectively
*-------------------------------------------------------------------------*/
      void   Register_As_Source(int );
      void   Register_As_Sink(int );


/*-------------------------------------------------------------------------*\
| these are volatile since they're accessed in read/write operations
| which can be called by process/scheduler thread
*-------------------------------------------------------------------------*/
      volatile Prty      ECOpriority;  // current priority of ECO
      volatile FiringMask   in_ports_ready; // binary code indicating ready i/p
ports

      volatile FiringMask   wakeup_call; // code that woke up the ECO
      volatile ProcessState process_state;


      Context ECOenv;

/*-------------------------------------------------------------------------*\
| base address of stack where this ECO process will
| reside. stacks for ECOs are actually allocated in heap
*-------------------------------------------------------------------------*/
      char*  stack;


      Stack_Size  stack_size;

      int wakeup_time;

};

#endif
```

```
/*=========================================================================*\
| ECO_I.cpp
|-------------------------------------------------------------------------
| Implementation file for ECO.
\*=========================================================================*/


#include "Q_Scalar_Data_Channels.h"
#include <iostream.h>

extern "C" void shallow_longjmp(Context* , int );
extern "C" int setjmp_all(Context* );


/*-------------------------------------------------------------------------*\
|   ECO Constructors
\*-------------------------------------------------------------------------*/
ECO :: ECO()
{
 inports_num=0;
 outports_num=0;
}

ECO :: ECO(Inports_Num in, Outports_Num on, FiringRule FR, Prty IP, Stack_Size
stk_size,
       Data_Channel** iports, Data_Channel** oports)
{
 inports_num = in;
 outports_num = on;
 F_Rule = FR;
 ECOpriority = IP;
 stack_size = stk_size;
 stack = new char[stack_size];

 process_state = wait_for_fire;

 inports = iports;
 outports = oports;

 for(int i=0; i<inports_num; i++)
   Register_As_Sink(i);

 for(int i=0; i<outports_num; i++)
   Register_As_Source(i);


 Register_OS();
}

/*-------------------------------------------------------------------------*\
|   Method to register process with kernel (in OS_All_Processes).
\*-------------------------------------------------------------------------*/
void ECO :: Register_OS()
{
 the_kernel.register_OS(this);
}
```

```
/*---------------------------------------------------------------------------*\
|   Method to register process as source of its output data channels.
\*---------------------------------------------------------------------------*/
void ECO :: Register_As_Source(int port_num)
{
 outports[port_num]->source_ECO=this;
 outports[port_num]->sourceport_num=port_num;
}


/*---------------------------------------------------------------------------*\
|   Method to register process as sink of its input data channels.
\*---------------------------------------------------------------------------*/
void ECO :: Register_As_Sink(int port_num)
{
 inports[port_num]->sink_ECO=this;
 inports[port_num]->sinkport_num=port_num;
}


/*---------------------------------------------------------------------------*\
|   Method that returns a pointer to the base of the stack in which the ECO
|runs.
\*---------------------------------------------------------------------------*/
unsigned int* ECO :: Stack()
{
 return ((unsigned int*) stack);
}


/*---------------------------------------------------------------------------*\
|   Method that retunrs the size of the stach in which the ECO runs.
\*---------------------------------------------------------------------------*/
Stack_Size ECO :: StackSize()
{
 return stack_size;
}
```

```
/*============================================================================*\
| Event_Queue.h
|-----------------------------------------------------------------------------
| Circular queue used to store 32-bit event codes corresponding to the events
| that have to be executed on receiving external interrupts.
\*============================================================================*/

#ifndef EVENT_QUEUE_H
#define EVENT_QUEUE_H 1

#include "ECO.h"

#define MAX_LENGTH_EVENT_Q   16


/*----------------------------------------------------------------------------*
 | Event Queue class - singleton
\*----------------------------------------------------------------------------*/
class Event_Queue
{
public:
    Event_Queue();

    ~Event_Queue();

    inline short int get_event(void)
    {
      if (front == rear) //queue empty
      {
        return -1;
      }
      else
      {
        front ++;

        if (front == MAX_LENGTH_EVENT_Q)
          front = 0;

        return event[front];
      }
    }

    inline short int Front()
    {
      return(front);
    }

    inline short int Rear()
    {
      return(rear);
    }
```

```
/*-------------------------------------------------------------------------*\
 | Method returns reference to a static event queue object - used every time
 | an event queue method has to be called.
\*-------------------------------------------------------------------------*/
    inline static Event_Queue& getInstance()
    {
       static Event_Queue the_event_queue;
       return the_event_queue;
    }


private:
    short int front;
    short int rear;
    short int event[MAX_LENGTH_EVENT_Q];
};

#endif



/*=========================================================================*\
| Event_Queue_I.cpp
|-------------------------------------------------------------------------
| Implementation file for Event Queue.
\*=========================================================================*/


#include "Event_Queue.h"


short int* ev_queue_rear_inx=0;
int ev_queue_max_len = MAX_LENGTH_EVENT_Q;


/*-------------------------------------------------------------------------*\
|  Constructor and destructor for Event_Queue
\*-------------------------------------------------------------------------*/


Event_Queue :: Event_Queue()
{
 front = 0; rear =-1;
 ev_queue_rear_inx = & rear;
}

Event_Queue :: ~Event_Queue()
{
 delete[] event;
}
```

```
/*=========================================================================*\
| Mailbox_Scalar_Data_Channel.h
|-------------------------------------------------------------------------
| Header file for Mailbox_Scalar_Data_Channel
\*=========================================================================*/

#ifndef MAILBOX_SCALAR_DATA_CHANNEL_H
#define MAILBOX_SCALAR_DATA_CHANNEL_H 1


#include "Data_Channel.h"


/*-------------------------------------------------------------------------*\
| Overflow handling for mailbox data channels
\*-------------------------------------------------------------------------*/
#if (DYNSCHD)
    #define BLOCK_ON_WRITE_MAILBOX(data_channel)\
    {\
        while (full(data_channel))           \
        {                           \
            switch (data_channel->overflow_style )        \
      {                       \
          case OS_Block:                \
                 data_channel->setBlocked();          \
                   Current->swap( blocked );               \
                 break;                   \
          case OS_Overwrite_Newest:              \
          case OS_Overwrite_Oldest:/* Both are same in case of mailboxes */ \
                      /* No need to do anything      */\
                      break; \
      }\
        }\
    }
#else
    #define BLOCK_ON_WRITE_MAILBOX(data_channel)
#endif



/*-------------------------------------------------------------------------*\
| Template class for mailbox data channels - takes the required data type that
| the channel is to store as parameter.
\*-------------------------------------------------------------------------*/
template<class Scalar_T> class Mailbox_Scalar_Data_Channel : public Data_Channel
{
 public:
      Mailbox_Scalar_Data_Channel(bool interrupt) : Data_Channel(sizeof(Scalar_T),
sizeof(Scalar_T), OS_Overwrite_Oldest, interrupt)
      {
        byte_dc = false;
      }
```

```
      Mailbox_Scalar_Data_Channel(enum Overflow_Style os, Cpcity cap, bool
interrupt) : Data_Channel(sizeof(Scalar_T), sizeof(Scalar_T),os, interrupt)
      {
        byte_dc = false;
      }

      ~Mailbox_Scalar_Data_Channel()
      {
      }

      inline void Read(Scalar_T& out_buf)
      {

        if(element_size <= size) // ASSERT
        {
          BLOCK_ON_READ(this);

          out_buf = *(Scalar_T*)(buffer + front);

          num_entries = 0;

          setmask_on_read();

          //switch to OS
          READ_GOTO_OS(interrupt_driven);
        }

        else
          cout << "ASSERT failed." << endl << flush;
      }

      inline void Write(Scalar_T in_buf)
      {

        BLOCK_ON_WRITE_MAILBOX(this);

        *(Scalar_T*) (buffer + rear ) = in_buf;

        num_entries = 1;

        setmask_on_write();


          // switch to OS once
        WRITE_GOTO_OS(interrupt_driven);
      }
};


#endif
```

```
/*=========================================================================*\
| Priority_Firing_Mask.h
|-------------------------------------------------------------------------
| Contains a firing mask and the priority the process will acquire when the
| corresponding firing mask is set. An array of these objects forms firing
| rule for an ECO.
\*=========================================================================*/

#ifndef PRIORITY_FIRING_MASK_H
#define PRIORITY_FIRING_MASK_H 1


typedef unsigned int FiringMask;
typedef short int  Prty;

class Priority_Firing_Mask
{
 public:

 Priority_Firing_Mask(FiringMask FM, Prty prty) :
             mask(FM), priority(prty)
 {
 }

 inline FiringMask Firing_Mask()
 {
   return mask;
 }

 inline Prty Priority()
 {
   return priority;
 }

/*-------------------------------------------------------------------------*\
| Method called during construction of a firing rule, following all other
| firing masks - serves as a delimiter.
\*-------------------------------------------------------------------------*/
 inline static Priority_Firing_Mask& done()
 {
   static Priority_Firing_Mask last(0x0, -1);
   return last;
 }

/*-------------------------------------------------------------------------*\
| Returns true if the firing mask is the delimiter (end of a firing rule has
| been reached)
\*-------------------------------------------------------------------------*/
 inline bool Last()
 {
   return(mask==0);
 }

private:
   FiringMask  mask;
   Prty     priority;
};
```

```
typedef Priority_Firing_Mask* FiringRule;

#endif


/*=========================================================================*\
| Q_Scalar_Data_Channel.h
|-------------------------------------------------------------------------
| Header file for Q_Scalar_Data_Channel
\*=========================================================================*/
#ifndef Q_SCALAR_DATA_CHANNEL_H
#define Q_SCALAR_DATA_CHANNEL_H 1


#include "Data_Channel.h"


/*-------------------------------------------------------------------------*\
| Overflow handling for queued data channels
\*-------------------------------------------------------------------------*/
#if (DYNSCHD)
    #define BLOCK_ON_WRITE_QUEUED(data_channel,length)\
    { \
    while(full(data_channel))\
    { \
            switch (data_channel->overflow_style) \
            { \
                case OS_Block  :  /*block*/ \
                                  data_channel->setBlocked(); \
                                  Current->swap( blocked ); \
                                  break; \
                case OS_Overwrite_Newest: /*re-adjust rear*/ \
                                       data_channel->rear -= length; \
                                       if (data_channel->rear < 0) \
                                       { \
                                          data_channel->rear += size; \
                                       } \
                      num_entries--; \
                      break; \
               case OS_Overwrite_Oldest: \
                                       data_channel->front += length; \
                                       if(data_channel->front == size) \
                                       { \
                                          data_channel->front = 0; \
                                       } \
                      num_entries--; \
                      break; \
         } \
       }\
    }
#else
    #define BLOCK_ON_WRITE_QUEUED(data_channel, length)
#endif
```

```
/*-------------------------------------------------------------------------*\
| Template class for queued data channels - takes the required data type that
| the channel is to store as parameter.
\*-------------------------------------------------------------------------*/
template<class Scalar_T> class Q_Scalar_Data_Channel : public Data_Channel
{

 public:
      Q_Scalar_Data_Channel(int size, bool interrupt) :
 Data_Channel(size*sizeof(Scalar_T), sizeof(Scalar_T), OS_Overwrite_Oldest,
interrupt)
      {
        byte_dc = false;
      }

      Q_Scalar_Data_Channel(int size, enum Overflow_Style os, Cpcity cap, bool
interrupt) : Data_Channel(size*sizeof(Scalar_T), sizeof(Scalar_T),os, interrupt)
      {
        byte_dc = false;
      }

      ~Q_Scalar_Data_Channel()
      {
      }


      inline void Read(Scalar_T& out_buf)
      {
        unsigned int i; //index

        if(element_size <= size) // ASSERT
        {
          BLOCK_ON_READ(this);

          out_buf = *(Scalar_T*)(buffer + front);
            front = ( front + element_size );

          if ( front == size)
          {
            front = 0;
          }

          num_entries--;

          setmask_on_read();

          //switch to OS
          READ_GOTO_OS(interrupt_driven);
        }

        else
          cout << "ASSERT failed." << endl << flush;
      }


      inline void Write(Scalar_T in_buf)
```

```
      {

        unsigned int i; //index
        int temp_rear;
        unsigned int rem_cap_buffer;


        if(element_size <= size)
        {
          BLOCK_ON_WRITE_QUEUED(this, element_size);

          *(Scalar_T*) (buffer + rear ) = in_buf;

          rear = ( rear + element_size );

          if ( rear == size)
          {
            rear = 0;
          }

          num_entries++;

          setmask_on_write();

          // switch to OS once
          WRITE_GOTO_OS(interrupt_driven);

        }
      }
};


#endif
```

```
/*============================================================================*\
| Ready_Queue.h
|-----------------------------------------------------------------------------
| Heap used to store references to ready processes - ordered by the process
| priorities. Used for dynamic scheduling.
\*============================================================================*/

#ifndef READY_QUEUE_H
#define READY_QUEUE_H 1


#include "ECO.h"
#include "Priority_Firing_Mask.h"
#include "typedefs.h"
#include <iostream.h>

class ECO;


/*----------------------------------------------------------------------------*
 | Ready queue class - singleton
\*----------------------------------------------------------------------------*/
#if (DYNSCHD)
class ready_queue
{
 private:

      ECO**  heap;
      int  size;

      int    capacity;

      inline void heapify(int i)
      {
        int k;
        int j=i;

        do
        {
          if(2*j + 1 <= i)
          {
            if( heap[2*j]->Current_Priority() < heap[2*j + 1]->Current_Priority()
)
              k=2*j;
            else
              k=2*j + 1;
          }
          else
            k = 2*j;
          if ( heap[j]->Current_Priority() > heap[k]->Current_Priority() )
          {
            ECO* temp = heap[k];
            heap[k] = heap[j];
            heap[j] = temp;
          }
          j*=2;
```

```
        } while(j<= size/2);
      }

 public:
      ready_queue();
      ready_queue(int );
      ~ready_queue();


/*-------------------------------------------------------------------------*\
 | Method returns reference to a static ready queue object - used every time
 | a ready queue method has to be called.
\*-------------------------------------------------------------------------*/
      inline static  ready_queue& getInstance()
      {
        static ready_queue Ready_Q;
        return Ready_Q;
      }


/*-------------------------------------------------------------------------*\
 | Returns pointer to the highest priority ready process - head of the queue.
\*-------------------------------------------------------------------------*/
      inline ECO* next_available_ECO()
      {
        return(heap[1]);
      }

      void bulk_insert(ECO *);
      void buildheap();


      inline void insert(ECO* process)
      {
          int i, new_i;

          i = ++size;
          new_i = i >> 1;  //i/2

        while ( i!=1
                  && process->Current_Priority() < heap[new_i]->Current_Priority())
        {
         heap[i] = heap[new_i];
         i = new_i;
         new_i >>= 1;
        }
        heap[i] = process;

      }

      inline ECO* remove_at_pos(int pos)
      {
        int i;

        ECO* process = heap[pos];

        for(i = pos; i<= size ; i++)
```

```
          heap[i-1] = heap[i];
        size--;

        for(i=size/2; i>=1; i--) heapify(i);

        return(process);
      }


      inline ECO* remove_topofheap()
      {
        ECO* process = remove_at_pos(1);
        return(process);
      }

      inline bool is_empty()
      {
        if(size == 0)    return(true);
        else      return(false);
      }


      inline bool is_full()
      {
        if(size == capacity)    return(true);
        else           return(false);
      }

      inline int  get_num_entries()
      {
        return(size);
      }


/*-------------------------------------------------------------------------*\
 | Method to change the priority of a given ECO to a specfied value and
 | reposition it in the heap.
\*-------------------------------------------------------------------------*/
      inline void change_value(ECO* eco, Prty new_priority)
      {
        int pos =1;

        //Finding the position of the process in the heap
        while (pos <= size)
        {
          if (heap[pos] == eco)
            break;
          else
            pos++;
        }

        ECO* process = heap[pos];
        remove_at_pos(pos);
        process->SetPriority(new_priority);
          insert(process);
      }
};
```

```
#endif

#endif


/*=========================================================================*\
| Ready_Queue_I.cpp
|-------------------------------------------------------------------------
| Implementation file for Ready Queue - needed only for the dynamically
| scheduled versions of the kernel.
\*=========================================================================*/

#include "Ready_Queue.h"
#include "DARKpp.h"
#include <iostream.h>

#if (DYNSCHD)

/*-------------------------------------------------------------------------*\
|  Constructor
\*-------------------------------------------------------------------------*/
ready_queue :: ready_queue()
{
 capacity = the_kernel.NumProcesses();
 heap = new ECO*[capacity+1];
}

/*-------------------------------------------------------------------------*\
|  Destructor
\*-------------------------------------------------------------------------*/
ready_queue :: ~ready_queue()
{
 delete[] heap;
}

/*-------------------------------------------------------------------------*\
|  (Optional) Method that can be used to add ECOs (not ordered by priority)
|iteratively.
\*-------------------------------------------------------------------------*/
void ready_queue :: bulk_insert(ECO *p)
{
 heap[++size] =p;
}

/*-------------------------------------------------------------------------*\
|  Method to be called if bulk_insert has been used to add ECOs.
\*-------------------------------------------------------------------------*/
void ready_queue :: buildheap()
{
 for(int i=size/2 ; i>=1 ; i--)
    heapify(i);
}

#endif
```

```
/*=========================================================================*\
 |  Timer.h
*-------------------------------------------------------------------------*
 |  We need to specify the time units for these OS calls.  We'd like for
 |  them to be as high-resolution as possible, but also portable.
\*=========================================================================*/
typedef unsigned int Time;  /* We need to pick units for this type.  */

typedef unsigned int Timer;

Time read_timer   ( Timer t );
void reset_timer  ( Timer t );
void set_timer    ( Timer t, Time new_time );
void set_timer_direction( Timer t, bool count_up );



/*=========================================================================*\
| Waiting_Queue.h
|-------------------------------------------------------------------------
| Circular queue used to store references to processes that are to be
| run after a user-specified delay has elapsed - ordered by the waiting times
\*=========================================================================*/

#ifndef WAITING_QUEUE_H
#define WAITING_QUEUE_H 1

#include <string.h>
#include "ECO.h"

#define MAX_LENGTH_WAITING_Q   16

class ECO;

/*-------------------------------------------------------------------------*
 |Waiting queue class - singleton
\*-------------------------------------------------------------------------*/
class Waiting_Queue
{
public:
    Waiting_Queue();

    ~Waiting_Queue();


/*-------------------------------------------------------------------------*
 |  Processes which are delayed are put into the waiting queue
\*-------------------------------------------------------------------------*/
    inline void insert(ECO* eco)
    {
       short int position = (front + 1) % MAX_LENGTH_WAITING_Q;

       //If the queue is empty
       if (front == rear)
       {
          rear = (rear + 1) % MAX_LENGTH_WAITING_Q;
```

```
        if (front == rear)
          ;//Queue full
        queue[rear] = eco;
        return;
      }


//Finding the position where to insert if the queue is not empty (earliest wakeup
        time)
      while(queue[position]->Wakeup_Time() <= eco->Wakeup_Time() && position !=
rear )
      {
        position = (position + 1) % MAX_LENGTH_WAITING_Q;
      }

      //If it is the last element
      if ( position == rear )
      {
        //the last element is smaller or equal ::: no need to shift
        if ( queue[position]->Wakeup_Time() <= eco->Wakeup_Time() )
        {
          rear = (rear + 1) % MAX_LENGTH_WAITING_Q;
          if (front == rear)
            ;//Queue full
          queue[rear] = eco;
          return;
        } //the last element is larger ::: shift
        else
        {
          rear = (rear + 1) % MAX_LENGTH_WAITING_Q;
          if (front == rear)
            ;//Queue full
          queue[rear] = queue[position];
          queue[position] = eco;
          return;
        }
      }
      else //element in between (Got the "position" where to insert)
      {
        short int temp_rear,temp_rear_previous = rear;
        rear = (rear + 1) % MAX_LENGTH_WAITING_Q;
        if (front == rear)
          ;//Queue full
        temp_rear = rear;
        while (temp_rear_previous != position)
        {
          queue[temp_rear] = queue[temp_rear_previous];
          temp_rear--; temp_rear_previous--;
          if (temp_rear < 0)
          {
            temp_rear = MAX_LENGTH_WAITING_Q - 1;
          }
          if (temp_rear_previous < 0)
          {
            temp_rear_previous = MAX_LENGTH_WAITING_Q - 1;
          }
        }
```

```
     queue[position] = eco;
     return;
   }
}

inline void Delete(ECO *eco)
{
   int pos = (front + 1) % MAX_LENGTH_WAITING_Q;
   if (front == rear)
     ;//Queue empty

   while (queue[pos] != eco)
   {
     pos = (pos + 1) % MAX_LENGTH_WAITING_Q;
   }

   //got the position
   if (pos <= rear) //single memcpy will do the job
   {
     memcpy(&queue[pos],&queue[(pos + 1) % MAX_LENGTH_WAITING_Q],rear-pos);
   }
   else //more memcpy required
   {
      memcpy(&queue[pos],&queue[(pos + 1) %
                              MAX_LENGTH_WAITING_Q],MAX_LENGTH_WAITING_Q-pos-1);
     queue[MAX_LENGTH_WAITING_Q-1] = queue[0];
     memcpy(&queue[0],&queue[1 % MAX_LENGTH_WAITING_Q],rear-1);
   }

   rear--;
   if (rear < 0)
   {
     rear = MAX_LENGTH_WAITING_Q-1;
   }
}

int get_earliest_time(void)
{
   if (front == rear) //queue empty
   {
     return -1;
   }
   else
   {
     return queue[(front + 1) % MAX_LENGTH_WAITING_Q]->Wakeup_Time();
   }
}


inline short int Front()
{
   return(front);
}

inline short int Rear()
{
   return(rear);
```

```
    }

/*-------------------------------------------------------------------------*
 | Method returns reference to a static waiting queue object - used every time
 | a waiting queue method has to be called.
\*-------------------------------------------------------------------------*/
    inline static Waiting_Queue& getInstance()
    {
       static Waiting_Queue the_waiting_queue;
       return the_waiting_queue;
    }


private:
    short int front;
    short int rear;
    ECO* queue[MAX_LENGTH_WAITING_Q];
};

#endif


/*=========================================================================*\
| Waiting_Queue_I.cpp
|-------------------------------------------------------------------------
| Implementation file for Waiting Queue.
\*=========================================================================*/

#include "Waiting_Queue.h"

/*-------------------------------------------------------------------------*\
|  Constructor and destructor for Waiting_Queue
\*-------------------------------------------------------------------------*/


Waiting_Queue :: Waiting_Queue()
{
 front = 0; rear =-1;
}


Waiting_Queue :: ~Waiting_Queue()
{
 delete[] queue;
}
```

```
/*=============================================================================*\
| convert_setjmp.asm
|-----------------------------------------------------------------------------
| Assembly code to save convert a shallow-saved thread context to a full save
| This is needed when the kernel is executing and an interrupt occurs causing
| a process of higher priority than the previously running process (whose
| context was saved partially) becomes ready. The context of the previously
| running process is saved entirely and the new ready process of higher
| priority is run.
\*=============================================================================*/

#include <asm_sprt.h>
#define dm_1 M6
#define dm_ptr I4
#define frame_ptr I6
#define stack_ptr I7
#define FETCH_RETURN I12=DM(M7, frame_ptr);
#define RETURN JUMP (M14,I12)
#define frame_stk DM(0,I6)
#define RESTORE_STACK stack_ptr=frame_ptr;
#define RESTORE_FRAME frame_ptr=frame_stk;


.segment/dm seg_dmda;
.var dmptr_store;
.endseg;


.segment/pm seg_pmco;
.global _convert_shallow_setjmp_to_all;

_convert_shallow_setjmp_to_all:

    push sts;          /* Save ASTAT and MODE1 registers  */
    DM(dmptr_store)=R4;     /* Point to Context structure     */
    dm_ptr=R4;

    MODE1=DM(dm_ptr,dm_1);    /* Restore alternate register mode */
    nop;

    dm_ptr = DM(dmptr_store);
    MODIFY(dm_ptr,1);       /* MODE1 already saved          */

    DM(dm_ptr,dm_1)=R1;     /* Saving 12 data registers       */
    DM(dm_ptr,dm_1)=R2;
    DM(dm_ptr,dm_1)=R3;
    DM(dm_ptr,dm_1)=R5;

    DM(dm_ptr,dm_1)=R6;
    DM(dm_ptr,dm_1)=R7;
    DM(dm_ptr,dm_1)=R9;
    DM(dm_ptr,dm_1)=R10;

    DM(dm_ptr,dm_1)=R11;
    DM(dm_ptr,dm_1)=R13;
    DM(dm_ptr,dm_1)=R14;
    DM(dm_ptr,dm_1)=R15;
```

```
/*Now save DAG1 */
R12=B0;
DM(dm_ptr,dm_1)=R12;
R12=I0;
DM(dm_ptr,dm_1)=R12;
R12=M0;
DM(dm_ptr,dm_1)=R12;
R12=L0;
DM(dm_ptr,dm_1)=R12;
R12=B1;
DM(dm_ptr,dm_1)=R12;
R12=I1;
DM(dm_ptr,dm_1)=R12;
R12=M1;
DM(dm_ptr,dm_1)=R12;
R12=L1;
DM(dm_ptr,dm_1)=R12;
R12=B2;
DM(dm_ptr,dm_1)=R12;
R12=I2;
DM(dm_ptr,dm_1)=R12;
R12=M2;
DM(dm_ptr,dm_1)=R12;
R12=L2;
DM(dm_ptr,dm_1)=R12;
R12=B3;
DM(dm_ptr,dm_1)=R12;
R12=I3;
DM(dm_ptr,dm_1)=R12;
R12=M3;
DM(dm_ptr,dm_1)=R12;
R12=L3;
DM(dm_ptr,dm_1)=R12;
R12=B5;
DM(dm_ptr,dm_1)=R12;
R12=I5;
DM(dm_ptr,dm_1)=R12;
R12=L5;
DM(dm_ptr,dm_1)=R12;
R12=B6;
DM(dm_ptr,dm_1)=R12;
R12=I6;
DM(dm_ptr,dm_1)=R12;
R12=L6;
DM(dm_ptr,dm_1)=R12;
R12=B7;
DM(dm_ptr,dm_1)=R12;
R12=I7;
DM(dm_ptr,dm_1)=R12;
R12=L7;
DM(dm_ptr,dm_1)=R12;

pop sts;
nop;

FETCH_RETURN
```

```
    RETURN (DB);       /* Back to caller */
    RESTORE_STACK
    RESTORE_FRAME
.ENDSEG;


/*=========================================================================*\
| longjmp_all.asm
|-------------------------------------------------------------------------
| Assembly code to restore ALL of the thread context. Needed for
| application-to-application context switching.
\*=========================================================================*/

#include <asm_sprt.h>
#define dm_1 M6
#define dm_ptr I4
#define frame_ptr I6
#define RETURN JUMP (M14,I12)
#define stack_ptr I7
#define RESTORE_STACK stack_ptr=frame_ptr;

.segment/dm seg_dmda;
.var return_value;
.var jmp_buf_ptr;
.endseg;

.segment/pm seg_pmco;
.global _longjmp_all;

_longjmp_all:       /* Most of the content taken from the longjmp.asm file */

 DM(return_value)=R8;
 DM(jmp_buf_ptr)=R4;
 dm_ptr=R4;

 MODE1=DM(dm_ptr,dm_1);NOP;

 dm_ptr=DM(jmp_buf_ptr);
 MODIFY(dm_ptr,1);
 R0=DM(return_value);
 IF EQ R0=R0+1;     /* Must never return 0 */

 R1=DM(dm_ptr,dm_1);   /*Begin restoring data regs */
 R2=DM(dm_ptr,dm_1);
 R3=DM(dm_ptr,dm_1);
 R5=DM(dm_ptr,dm_1);

 R6=DM(dm_ptr,dm_1);
 R7=DM(dm_ptr,dm_1);
 R9=DM(dm_ptr,dm_1);
 R10=DM(dm_ptr,dm_1);

 R11=DM(dm_ptr,dm_1);
 R13=DM(dm_ptr,dm_1);
 R14=DM(dm_ptr,dm_1);
 R15=DM(dm_ptr,dm_1);
```

```
 /* We can read directly into DAG1 as long as we use
  * preindexed, not postindexed addressing */

 B0=DM(0,dm_ptr);
 I0=DM(1,dm_ptr);
 M0=DM(2,dm_ptr);
 L0=DM(3,dm_ptr);

 B1=DM(4,dm_ptr);
 I1=DM(5,dm_ptr);
 M1=DM(6,dm_ptr);
 L1=DM(7,dm_ptr);

 B2=DM(8,dm_ptr);
 I2=DM(9,dm_ptr);
 M2=DM(10,dm_ptr);
 L2=DM(11,dm_ptr);

 B3=DM(12,dm_ptr);
 I3=DM(13,dm_ptr);
 M3=DM(14,dm_ptr);
 L3=DM(15,dm_ptr);

 B5=DM(16,dm_ptr);
 I5=DM(17,dm_ptr);
 L5=DM(18,dm_ptr);

 B6=DM(19,dm_ptr);
 I6=DM(20,dm_ptr);
 L6=DM(21,dm_ptr);

 B7=DM(22,dm_ptr);
 I7=DM(23,dm_ptr);
 L7=DM(24,dm_ptr);


 MODIFY(dm_ptr, 25);/* Advance to Special Register area */

 PCSTKP=DM(dm_ptr,dm_1);NOP;     /* Unwind call stack */
 LADDR=DM(dm_ptr,dm_1);       /* Unwind loop stack */

 I12=DM(dm_ptr,dm_1);       /* Should be return address */

 RETURN (DB);
 RESTORE_STACK
 frame_ptr = R2;  /* The call to setjmp would have put
           the frame pointer in R2, and we've restored
           that value, so we can recover the frame
           pointer this way. */
.endseg;
```

```
/*=============================================================================*\
| setjmp_all.asm
|-----------------------------------------------------------------------------
| Assembly code to save ALL of the thread context. Needed for
| application-to-application context switching.
\*=============================================================================*/

#include <asm_sprt.h>
#define dm_1 M6
#define dm_ptr I4
#define frame_ptr I6
#define stack_ptr I7
#define FETCH_RETURN I12=DM(M7, frame_ptr);
#define RETURN JUMP (M14,I12)
#define frame_stk DM(0,I6)
#define RESTORE_STACK stack_ptr=frame_ptr;
#define RESTORE_FRAME frame_ptr=frame_stk;



.segment/pm seg_pmco;
.global _setjmp_all;

_setjmp_all:

    dm_ptr=R4;         /*Point to Context structure */

    DM(dm_ptr,dm_1)=MODE1;    /* Save MODE1 */

    DM(dm_ptr,dm_1)=R1;   /*Saving 12 data registers  */
    DM(dm_ptr,dm_1)=R2;
    DM(dm_ptr,dm_1)=R3;
    DM(dm_ptr,dm_1)=R5;

    DM(dm_ptr,dm_1)=R6;
    DM(dm_ptr,dm_1)=R7;
    DM(dm_ptr,dm_1)=R9;
    DM(dm_ptr,dm_1)=R10;

    DM(dm_ptr,dm_1)=R11;
    DM(dm_ptr,dm_1)=R13;
    DM(dm_ptr,dm_1)=R14;
    DM(dm_ptr,dm_1)=R15;

    R12=B0;
    DM(dm_ptr,dm_1)=R12;
    R12=I0;
    DM(dm_ptr,dm_1)=R12;
    R12=M0;
    DM(dm_ptr,dm_1)=R12;
    R12=L0;
    DM(dm_ptr,dm_1)=R12;
    R12=B1;
    DM(dm_ptr,dm_1)=R12;
    R12=I1;
    DM(dm_ptr,dm_1)=R12;
```

```
    R12=M1;
    DM(dm_ptr,dm_1)=R12;
    R12=L1;
    DM(dm_ptr,dm_1)=R12;
    R12=B2;
    DM(dm_ptr,dm_1)=R12;
    R12=I2;
    DM(dm_ptr,dm_1)=R12;
    R12=M2;
    DM(dm_ptr,dm_1)=R12;
    R12=L2;
    DM(dm_ptr,dm_1)=R12;
    R12=B3;
    DM(dm_ptr,dm_1)=R12;
    R12=I3;
    DM(dm_ptr,dm_1)=R12;
    R12=M3;
    DM(dm_ptr,dm_1)=R12;
    R12=L3;
    DM(dm_ptr,dm_1)=R12;
    R12=B5;
    DM(dm_ptr,dm_1)=R12;
    R12=I5;
    DM(dm_ptr,dm_1)=R12;
    R12=L5;
    DM(dm_ptr,dm_1)=R12;
    R12=B6;
    DM(dm_ptr,dm_1)=R12;
    R12=I6;
    DM(dm_ptr,dm_1)=R12;
    R12=L6;
    DM(dm_ptr,dm_1)=R12;
    R12=B7;
    DM(dm_ptr,dm_1)=R12;
    R12=I7;
    DM(dm_ptr,dm_1)=R12;
    R12=L7;
    DM(dm_ptr,dm_1)=R12;

    /* Now save hardware specific state */

        DM(dm_ptr,dm_1)=PCSTKP;         /* Save current HW SP    */
        NOP;                            /* One cycle latency     */
    DM(dm_ptr,dm_1)=LADDR;       /* Save Loop stack depth */

    R0=R0-R0, FETCH_RETURN       /* Return 0 for setjmp   */
    DM(dm_ptr,dm_1)=I12;      /* Save return address   */

restore_state:  RETURN (DB);       /* Back to caller     */
    RESTORE_STACK
    RESTORE_FRAME
.ENDSEG;
```

```
/*=============================================================================*\
| setjmp_initialize.asm
|------------------------------------------------------------------------------
| Assembly code used for initializing process context information
\*=============================================================================*/


#include <asm_sprt.h>
#define dm_1 M6
#define dm_ptr I4
#define frame_ptr I6
#define stack_ptr I7
#define FETCH_RETURN I12=DM(M7, frame_ptr);
#define RETURN JUMP (M14,I12)
#define frame_stk DM(0,I6)
#define RESTORE_STACK stack_ptr=frame_ptr;
#define RESTORE_FRAME frame_ptr=frame_stk;


.segment/pm seg_pmco;
.global _setjmp_initialize;

_setjmp_initialize:

    dm_ptr=R4;         /*Point to Context structure */

    /* Flip the register set to change MODE for applications*/
    R12 = 0x000004F8;
    R8 = MODE1;
    R12 = R12 or R8;
    DM(dm_ptr,dm_1)=R12;  /* Save MODE1 */

    DM(dm_ptr,dm_1)=R1;   /*Saving 12 data registers  */
    DM(dm_ptr,dm_1)=R2;
    DM(dm_ptr,dm_1)=R3;
    DM(dm_ptr,dm_1)=R5;

    DM(dm_ptr,dm_1)=R6;
    DM(dm_ptr,dm_1)=R7;
    DM(dm_ptr,dm_1)=R9;
    DM(dm_ptr,dm_1)=R10;

    DM(dm_ptr,dm_1)=R11;
    DM(dm_ptr,dm_1)=R13;
    DM(dm_ptr,dm_1)=R14;
    DM(dm_ptr,dm_1)=R15;

    /*Now save DAG1 */
    R12=B0;
    DM(dm_ptr,dm_1)=R12;
    R12=I0;
    DM(dm_ptr,dm_1)=R12;
    R12=M0;
    DM(dm_ptr,dm_1)=R12;
    R12=L0;
    DM(dm_ptr,dm_1)=R12;
    R12=B1;
```

```
    DM(dm_ptr,dm_1)=R12;
    R12=I1;
    DM(dm_ptr,dm_1)=R12;
    R12=M1;
    DM(dm_ptr,dm_1)=R12;
    R12=L1;
    DM(dm_ptr,dm_1)=R12;
    R12=B2;
    DM(dm_ptr,dm_1)=R12;
    R12=I2;
    DM(dm_ptr,dm_1)=R12;
    R12=M2;
    DM(dm_ptr,dm_1)=R12;
    R12=L2;
    DM(dm_ptr,dm_1)=R12;
    R12=B3;
    DM(dm_ptr,dm_1)=R12;
    R12=I3;
    DM(dm_ptr,dm_1)=R12;
    R12=M3;
    DM(dm_ptr,dm_1)=R12;
    R12=L3;
    DM(dm_ptr,dm_1)=R12;
    R12=B5;
    DM(dm_ptr,dm_1)=R12;
    R12=I5;
    DM(dm_ptr,dm_1)=R12;
    R12=L5;
    DM(dm_ptr,dm_1)=R12;
    R12=B6;
    DM(dm_ptr,dm_1)=R12;
    R12=I6;
    DM(dm_ptr,dm_1)=R12;
    R12=L6;
    DM(dm_ptr,dm_1)=R12;
    R12=B7;
    DM(dm_ptr,dm_1)=R12;
    R12=I7;
    DM(dm_ptr,dm_1)=R12;
    R12=L7;
    DM(dm_ptr,dm_1)=R12;

    /* Now save hardware specific state */

        DM(dm_ptr,dm_1)=PCSTKP;         /* Save current HW SP */
        NOP;                            /*One cycle latency*/
    DM(dm_ptr,dm_1)=LADDR;     /* Save Loop stack depth */

    R0=R0-R0, FETCH_RETURN     /*Return 0 for setjmp*/
    DM(dm_ptr,dm_1)=I12;    /* Save return address */

restore_state:  RETURN (DB);       /* Back to caller */
    RESTORE_STACK
    RESTORE_FRAME
.ENDSEG;
```

```
/*=========================================================================*\
| shallow_longjmp.asm
|-------------------------------------------------------------------------
| Assembly code to partially restore the registers, used when register set is
| changed. Used in application-to-kernel/kernel-to-applcation context switching
\*=========================================================================*/

#include <asm_sprt.h>
#define dm_1 M6
#define dm_ptr I4
#define frame_ptr I6
#define stack_ptr I7
#define RESTORE_STACK stack_ptr=frame_ptr;
#define RETURN JUMP (M14,I12)

.segment/dm seg_dmda;
.var dmptr_store;
.var return_val;
.endseg;

.segment/pm seg_pmco;
.global _shallow_longjmp;


_shallow_longjmp:

 R0=PASS R8, dm_ptr=R4;     /*Point to jmp_buf*/
 IF EQ R0=R0+1;     /* Must never return 0 */
 DM(dmptr_store)=R4;
 DM(return_val)=R0;

 MODE1=DM(dm_ptr,dm_1);     /* Restore mode */
 nop;

 R0=DM(return_val);
 dm_ptr=DM(dmptr_store);
 MODIFY(dm_ptr,38);      /* Already restored the MODE1 and jumping forward in the
context structure */

 PCSTKP=DM(dm_ptr,dm_1);NOP;     /* Unwind call stack */
 LADDR=DM(dm_ptr,dm_1);     /* Unwind loop stack */

 I12=DM(dm_ptr,dm_1);    /* Should be return address */

 restore_state: RETURN (DB);
 RESTORE_STACK
 frame_ptr = R2;  /* The call to setjmp would have put
           the frame pointer in R2, and we've restored
           that value, so we can recover the frame
           pointer this way. */
.ENDSEG;
```

```
/*=============================================================================*\
| shallow_setjmp.asm
|------------------------------------------------------------------------------
| Assembly code to partially save the registers, used when register set is
| changed. Used in application-to-kernel/kernel-to-applcation context switching
\*=============================================================================*/

#include <asm_sprt.h>
#define dm_1 M6
#define dm_ptr I4
#define frame_ptr I6
#define stack_ptr I7
#define frame_stk DM(0,I6)
#define RESTORE_STACK stack_ptr=frame_ptr;
#define RESTORE_FRAME frame_ptr=frame_stk;
#define RETURN JUMP (M14,I12)
#define FETCH_RETURN I12=DM(M7, frame_ptr);

.segment/pm seg_pmco;
.global _shallow_setjmp;


_shallow_setjmp:

 dm_ptr=R4;       /*Point to jmp_buf*/

 DM(dm_ptr,dm_1)=MODE1;       /* Save MODE1 */

 MODIFY(dm_ptr,37);

 DM(dm_ptr,dm_1)=PCSTKP;         /* Save current HW SP */
    NOP;                            /*One cycle latency*/
 DM(dm_ptr,dm_1)=LADDR;     /* Save Loop stack depth */

 R0=R0-R0, FETCH_RETURN     /*Return 0 for setjmp*/
 DM(dm_ptr,dm_1)=I12;    /* Save return address */

 restore_state: RETURN (DB);  /*    Back to caller */
 RESTORE_STACK
 RESTORE_FRAME

.ENDSEG;
```

Vita

Sumithra Bhakthavatsalam was born in Chennai, India on September 25th 1977. She moved to Bangalore at the age of four and spent the next twenty years of her life there.

After securing a B.E (Bachelor of Engineering) degree in Computer Science & Engineering from Bangalore University, she worked as a Software Engineer for a period of one year in Infosys Technologies Ltd., Electronics City, Bangalore, an SEI CMM level V company.

After this she joined Virginia Polytechnic Institute & State University in Spring 2001 for a Master's degree in Computer Science & Applications and graduated in May 2003. Her interests lie in Operating Systems, Software for Embedded Systems and Programming Languages.