INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

THE UNIVERSITY OF CALGARY

A Metamorphic Control Architecture for Holonic Systems

by

Sivaram Balasubramanian

A DISSERTATION

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF MECHANICAL ENGINEERING

CALGARY, ALBERTA

SEPTEMBER, 1997

Abstract

Next generation manufacturing systems are envisioned to be comprised of distributed network of autonomous and cooperative holonic resources. Holonic systems have an information processing part and a physical processing part to process knowledge and material simultaneously. Holonic systems are evolutionary in nature to better accomplish current system objectives. In other words the form and substance of holonic systems undergo constant transformation (metamorphosis) through out their life time. Real time control of such holonic systems requires a radically different approach from that of traditional unit level regulatory control systems.

The control requirements of holonic systems are distributed in nature. The dynamics of distributed control results in complex system behavior and requires an event driven control system. Since holonic systems are evolutionary in nature, they require both static and dynamic reconfigurability of their control systems. Additionally, holonic systems require incorporation of intelligence into their control systems that can enhance autonomy and cooperation. The conventional centralized scan based control systems are inadequate in meeting the said control requirements of holonic systems. This necessitates a new and novel system level distributed control approach, and the control systems based on this approach are termed as metamorphic control systems.

The engineering of such software centric metamorphic control systems for dynamically reconfigurable distributed multi-sensor based holonic systems, is addressed in this dissertation. An integrated and uniform event driven control architecture is specified for various functional levels of metamorphic control system. The architecture utilizes the emerging International Electrotechnical Commission function block standard (IEC 1499) for industrial process measurement and control systems, to specify the requisite behavior of distributed control software components (agents).

A prototype metamorphic control system has been developed using the new architecture. The core metamorphic control mechanisms have been developed in the form

of a distributed real time operating system. The function block specification is used to develop distributed control software agents and applications. A system engineering interface has been developed for remote program development, configuration and maintenance of distributed control system. The implementation and evaluation details of this prototypical system are presented.

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Douglas H. Norrie, for his direction and advice through out this research. I especially appreciate the freedom and flexibility, that led me to explore a relatively new area of research. I also appreciate his efforts for making sure that lack of resources never impeded progress.

I would like to thank my supervisory committee members, Dr. Paul Rogers and Dr. Keith Chrystall, for their suggestions, time and energy during the course of this work.

Special thanks are extended to IEC 1499 Function Block Standards Committee project leader, Dr. James Christensen, for giving me 'inside' information. I gained a lot from the discussions with him and am grateful to him for lending an ear.

I thank our technical supervisor, Nick Vogt, for his help in obtaining equipment and other resources for this research.

I am grateful to our technician, David Genge, for his help in developing special hardware used in this research.

I am also thankful to our graduate secretary, Lynn Banach, and division secretary, Karen Undseth, for their help during this research.

I am indebted to my colleagues, Harish Ananda Rao and Francisco Paul Maturana for their help and invigorating discussions.

I am also privileged to have many good friends who have helped me in one way or the other. Thanks to you all.

Words cannot describe my gratitude adequately, to my beloved parents and family members for their encouragement and support, without which this endeavor would not have been possible.

Dedicated to my beloved parents.

Table of Contents

List of Tables

List of Figures

Acronyms

API	Application Programming Interface
CNC	Computer Numerical Control
CPU	Central Processing Unit
DCOS	Distributed Controller Operating System
DCS	Distributed Control System
ECC	Execution Control Chart
FIFO/LIFO	First In First Out/Last In First Out
I/O	Input/Output
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
ISA/EISA	Industry Standard Architecture/Extended Industry Standard Architecture
ISO	International Standards Organization
LAN	Local Area Network
MAP	Manufacturing Automation Protocol
MB/GB	Mega Byte/Giga Byte
MMS	Manufacturing Message Specification
NC	Numerical Control
OAC	Open Architecture Control
OS	Operating System
OSI	Open Systems Interconnection
PC	Personal Computer
PCI	Peripheral Component Interconnect
PLC	Programmable Logic Controller
TCP/UDP/IP	Transmission Control Protocol/User Datagram Protocol/Internet Protocol

Glossary

Agent	An active control software component/module/object of a holon.
Autonomy	The capability of an entity to create and control the execution of its own plans and/or strategies.
Cooperation	A Process whereby a set of entities develop mutually acceptable plans and execute them.
Function Block	A software functional unit comprising an individual. named copy of a data structure and associated operations.
Holarchy	A system of holons which can cooperate to achieve a goal or objective. The holarchy defines the basic rules for cooperation of the holons and thereby limits their autonomy.
Holon	An autonomous and cooperative building block of a manufacturing system for transforming, transporting. storing and/or validating information and physical objects. The holon consists of an information processing part and often a physical processing part. A holon can form part of another holon.
Interoperability	The ability of an entity to cooperate with other dissimilar entities. Also the ability of heterogeneous hardware and software subsystems to function together.
Loosely Coupled	A distributed multi-processor system with no shared primary memory.
Metamorphic	A control system that undergoes constant transformation of form and substance through out its life time.
Tightly Coupled	A multi-processor system with shared primary memory.

Chapter 1

Introduction

1.1 Overview

Computer control of manufacturing systems has been the focus of extensive research over the last several decades. Advances in microprocessor, computing, networking and interfacing technologies have improved the capabilities of industrial automation and control systems substantially over this period. However, these control systems are proprietary and still have problems in areas such as interoperability, scalability, upgradability (without complete replacement), and lack of standard interfaces. The development of open architecture control systems addresses some of these problems, in varying degrees. Open architecture control systems shift the focus of automation from being hardware centric to software centric, providing further flexibility. The focus is now shifting to distributed control systems which is the central concern of this dissertation research work.

This dissertation addresses the engineering of software centric control systems for cooperating networks of distributed autonomous sub-systems to provide for enhanced interoperability, scalability, and upgradability. A comprehensive control architecture is presented for dynamically reconfigurable distributed multi-sensor based systems. A prototype control framework has been developed using a system level approach. The core distributed control mechanisms have been developed in the form of a distributed real time operating system. A state machine based sophisticated application specification model is used to develop reusable software modules. A system engineering interface has been developed to address the configuration and maintenance requirements of the distributed control system.

In this chapter, the concepts of holonic manufacturing paradigm and its Intelligent Control System are introduced. Subsequently, the motivation for developing a new control

architecture, and the objectives of this dissertation are presented. Finally, the organizational structure of the remaining chapters in this dissertation is outlined.

1.2 Holonic Systems

The change in market requirements towards a larger variety of products in smaller batch sizes, has lead to the concept of next generation intelligent manufacturing systems being an integrated network of distributed resources simultaneously capable of combined knowledge processing and material processing [Norrie94]. The control relationships among these distributed resources need to be reconfigured "on the fly" according to changing requirements [Norrie94]. Earlier research in the area of intelligent manufacturing systems has established that such resources can be realized through the concepts of holonic paradigm [Chris94a].

Arthur Koestler [Koes71] established the basic concepts of holonic systems by postulating a set of underlying principles to explain the self organising tendencies of social and biological systems. He proposed the term *holon* to describe the building blocks of these systems. This is a combination of the Greek word *holos*, meaning "whole", with the suffix *-on* meaning "part". This term reflects the ability of holons to act as autonomous entities, yet cooperating to form self-organizing hierarchies of subsystems. Koestler used the term *holarchy* to describe these holonic hierarchies. These concepts have been subsequently extended and applied in the context of manufacturing systems by an international consortium of industry and academia on Holonic Manufacturing Systems (HMS) [Chris94a].

In the context of manufacturing systems, a holon is defined as an autonomous and cooperative building block of a manufacturing system for transforming, transporting, storing and/or validating information and physical objects [Chris94b]. A holon has the autonomy to create and control the execution of its own plans, and can cooperate with other holons to develop mutually acceptable plans for achieving system goals. Cooperation among holons is accomplished through an evolutionary self-organizing holarchy. A holon consists of an information processing part and a physical processing

part, and can form a part of another holon.

Fig. 1.1 illustrates the major functional elements and critical interfaces of a holon. As may be noted, the functional elements of holon are modular in nature and the architecture is highly distributed. Further, the constitution of holons change and the functional elements evolve over time according to system level requirements. For instance, a turning center holon may be augmented with certain milling operations or a holonic robot may be provided with additional vision system capabilities. Hence the ability to reconfigure on demand is an important requirement for holonic systems. This and other capabilities of holonic systems are largely dependent on their Intelligent Control Systems.

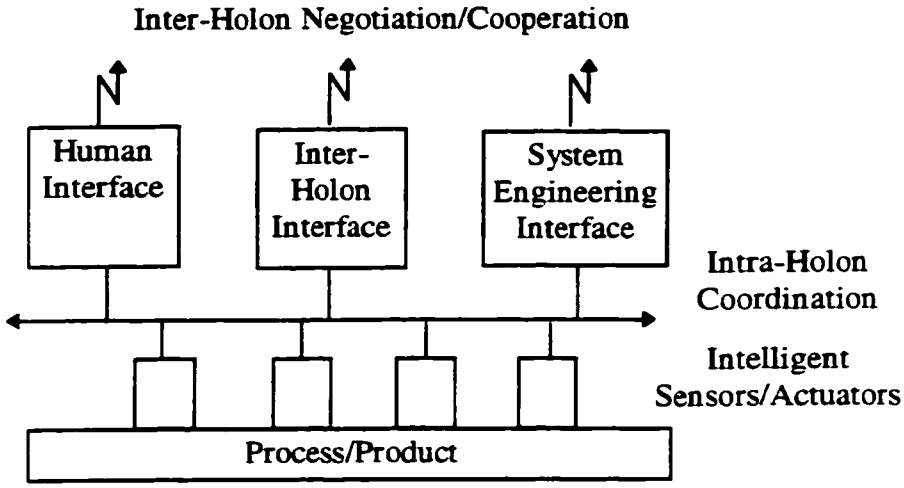


Figure 1.1: Holonic Elements and Interfaces

As shown in Fig. 1.2, the Intelligent Control System has been identified [Chris94b] to be comprised of four major components:

- The Process/Machine Control block, responsible for execution of the control plan for the process being controlled.
- The Process/Machine Interface block, representing the physical and logical interface (sensors and actuators) to the process being controlled.
- The Human Interface block, representing the interfaces to the human resources.
- The Inter-Holon Interface block, which provides for the exchange of information, negotiation and cooperation, with other holons in the system.

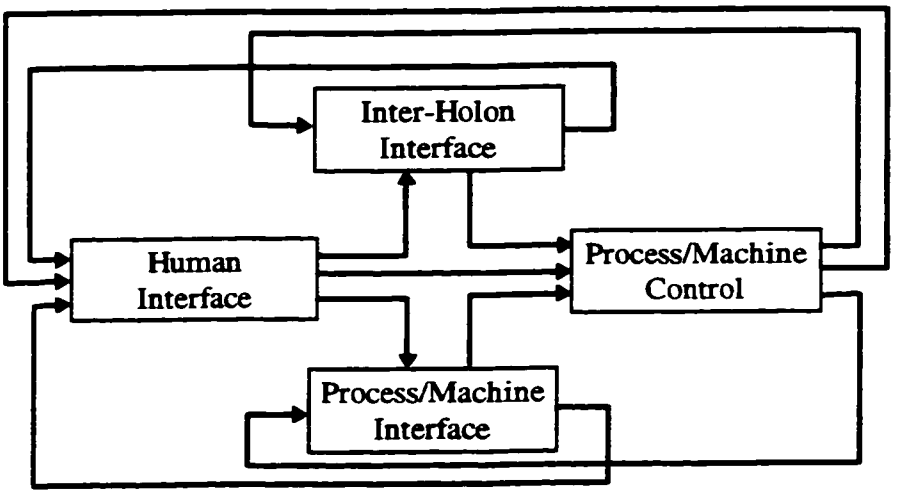


Figure 1.2: Intelligent Control System

In this dissertation, the architecture and functional elements of these four major components are developed. As mentioned earlier, the form and substance of holonic systems, and the control relationships among them, evolves continuously according to changing requirements. Due to this constant transformation of a holonic system within its lifetime, its control system undergoes metamorphosis. Hence we term this control system as "metamorphic". The issues associated with metamorphic control of holonic systems include predictable real time performance under distributed computation and control, dynamic reconfiguration and fault tolerance, distributed machine intelligence and cooperation, and distributed system engineering interface. The architectural requirements and development of such control systems have not been addressed before, in an integrated manner.

1.3 Motivation

Real world autonomous and cooperative systems such as holonic machines operate under real-time constraints and are inherently distributed and dynamic. Traditional system structures based on static and hierarchical control do not suffice for such situations, and hence a new generation of control systems are needed. Modern and future machinery will therefore, include embedded modular distributed real-time computer control systems. The control system is distributed since computer nodes are spatially distributed in the machine

and control functions are distributed over the nodes. Nodes acquire sensor data, perform processing, exchange information over a real-time network, synchronize, and perform actuation, in order to achieve the system goals. Distributed control systems such as these can provide improved functionality, performance, flexibility, and reduced complexity and costs.

In order to function effectively, the distributed hardware and software systems which monitor and control real world processes must provide adequate means to cope with application requirements such as timeliness, concurrency, and decentralization. A predictable real time communication protocol is the backbone of a distributed control system. In addition, the control system also has to provide high degree of robustness and fault-tolerant behavior through the use of techniques such as component redundancies, dynamic reconfiguration mechanisms, and distributed intelligent sensors and actuators.

Real-time computer control systems arose and have evolved primarily in the context of small, simple, static, centralized, subsystems for unit-level, sampled data monitoring and regulatory control. Holonic systems move from centralized to distributed control, because the problem is distributed, and because of flexibility and cost-effectiveness. However there is a price to pay: system development becomes more complex than in the case of centralized control. There are no comprehensive methods or tools currently available to develop distributed control systems for holonic systems. Today, there are many commercial components that are available "off the shelf" for standalone control systems. Unfortunately, these systems do not scale up beyond the unit level and are unable to meet the needs of future applications in respect to performance, reliability, and extendibility.

For the new distributed control systems, a new design step is introduced in which the control tasks need to be structured and partitioned such that they lend themselves to distributed allocation. There are no tools to support this step and evaluate the effect on control system performance due to different allocations. In a real-time control system, several modes of operation usually need to be implemented as well as mode transitions among them. Data processing and communication in each case pose different requirements

with respect to delays, consistency, and error detection and handling. Existing systems do not provide any support to achieve these requirements nor for a host of others. In short, it becomes the responsibility of system designer/developer to meet any shortcoming at system level through a "piece meal" approach at the application level.

This lack of architecture and means for developing dynamically reconfigurable distributed control systems provides the prime motivation for the work presented in this dissertation. Having outlined the motivation, the following section describes the objectives of this dissertation.

1.4 Objectives

The goal of this dissertation is to provide a comprehensive framework for engineering dynamically reconfigurable distributed control of multi-sensor-based systems. It is targeted towards improving the capabilities, reliability and performance of distributed control systems, while at the same time significantly reducing the development time and costs. To achieve this goal, the research has focused on the following objectives:

- To identify the architectural components for generic metamorphic control of holonic systems.
- To develop the core distributed control mechanisms in the form of a distributed real time operating system.
- To develop a consistent programming model and the associated libraries for mapping application level requirements.
- To develop a graphical system engineering interface for configuration and maintenance of the distributed control system.

1.5 Organization of Dissertation

Having provided the motivation and objectives, this section outlines the contents of this dissertation. In Chapter 2, the state of the art in industrial automation and control is discussed. This is followed by an overview of emerging open architecture control technology. In Chapter 3, the requirements for metamorphic control of holonic systems

are identified and the need for a new and novel system-level approach is demonstrated. This is followed by a review of agent based intelligent control architectures. In Chapter 4, the relevant literature in the areas of distributed real time operating systems and formal specification methods of distributed real time systems, are reviewed.

In Chapter 5, a novel agent based metamorphic control architecture is presented and the critical components and issues of this architecture are identified. In Chapter 6, the concepts and architecture of a new distributed real time operating system is presented. In Chapter 7, the design and implementation of this distributed operating system are described. In Chapter 8, one feasible method for developing application software and a system engineering interface for developing application software, configuration and maintenance, are presented. In Chapter 9, the details of implementation infrastructure and evaluation of the implemented system are presented. Finally in Chapter 10, the conclusions are summarized, and the anticipated contributions and areas for future research are identified.

Chapter 2

Industrial Automation and Control

2.1 Introduction

In an automated manufacturing system, the objective is to achieve a complete spectrum of manufacturing control functions ranging from production planning and control at the highest level, to process/machine control at the lowest level. Intelligent manufacturing involves not only the achievement of these control functions, but seamless integration of these as well. Traditionally, these objectives have been achieved by horizontal integration across an hierarchy of control layers. However, for an autonomous holon the control objectives of traditional layers are partitioned vertically. Hence, a holon-based decentralized and distributed manufacturing system uses cooperation as the primary means to achieve system wide integration of control functions. In other words, the control objectives which need to be achieved at the individual holon level need also to be achieved at the system level. Obviously, the ability of a holon to achieve these control functions is directly dependent on its control system.

This chapter begins by presenting an overview of existing industrial control systems technology and is followed by a discussion on factory floor communications standards that are crucial for system integration. The drawbacks of extant control systems technology have led to a number of software centric open architecture control initiatives that are destined to impact the future of the control industry. Therefore, these efforts are reviewed in detail.

2.2 Industrial Control Systems

An industrial process/machine control system is the "sense and brain for the muscle" behind any automated manufacturing equipment. Its function is that of periodically comparing sensory process/discrete input variables with setpoints/logic states,

computing the outputs according to a predefined control algorithm/logic and communicating the output signals to the final control element for actuation. The industrial process/machine control systems used in a manufacturing scenario can be classified into following categories: Programmable Logic Control systems, Process/Loop Control systems, Distributed Control Systems, Computer Numerical Control systems and Robot Control systems. The following sub-sections discuss briefly the state of the art in these control systems.

2.2.1 Programmable Logic Control Systems

The first Programmable Logic Control (PLC) [Bryan88] system was introduced in the late sixties as a replacement for massive hard-wired relay panels then used in manufacturing plants. In addition to the key feature of programmability, PLCs provided modularity, expandability, diagnosis indication and reliability under extreme factory floor operating conditions. Although originally designed for on/off applications, such as controlling the starting and stopping of transfer lines, PLCs rapidly spread to more sophisticated applications, such as those in the process industries. The evolution of the PLC over the years, due primarily to advancements in microprocessors, high-speed communication networks and software has gained it a central place in industrial automation.

The architecture of a PLC resembles that of a general purpose microcomputer and can be considered representative of other controller systems as well. Basic PLC components are designed as self-contained modularized units that can be inserted and removed from industrial racks or panels. As shown in Fig. 2.1, the hardware platform of a PLC has three main sub-systems: a system power supply module, a processor module and Input/Output (I/O) interface modules. The PLC may also include a peripheral programming device and an interface to a data communications network. The system power supply module provides the necessary voltages for the correct operation of primary PLC components and also usually includes a battery backup to provide power to memory in case of a power failure.

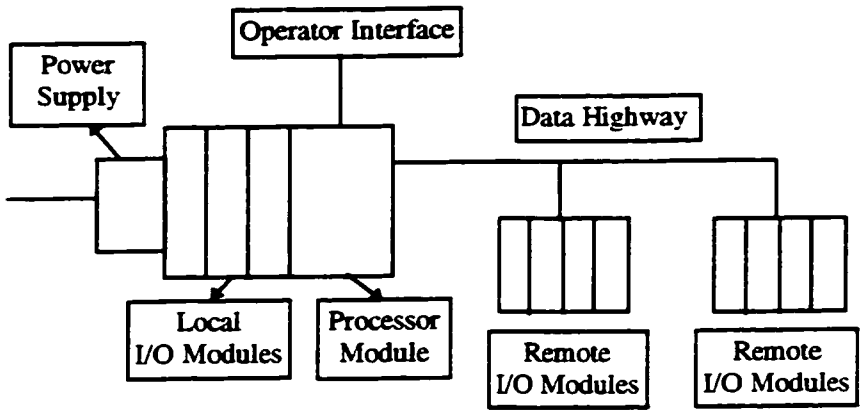


Figure 2.1: Programmable Logic Control System

The processor module houses one or more microprocessors, their supporting circuitry and memory. The processor module also includes diagnostic indicators designed to detect communication failures as well as other failures during system operation. The memory is partitioned into two regions: a system memory region and an application memory region. The system memory includes an area called the executive or operating system, composed of a collection of permanently stored programs that direct all system activities, such as execution of user control programs, communication with peripheral devices etc. The system memory also contains the routines that implement the PLC instruction set, which is composed of specific control functions such as logic, sequencing, timing, counting and arithmetic.

The application memory is divided into the data table area and user program area. The data table stores any data associated with the user control program, such as system input/output status data, constants, variables, preset values etc. The data table is where data is monitored, manipulated and changed for control purposes. The user program area is where the programmed instructions entered by the user are stored as an application control program.

The I/O interface modules connects a PLC to external factory floor sensor/actuator field devices. The main purpose of the I/O interface is to condition the various signals received from or sent to the external input and output devices. Input and output modules are housed in the same master rack or panel that houses the other

components of PLC system. Additional I/O modules can be housed in a remote I/O racks that communicate with the master rack about I/O and diagnostic information. Every I/O module has its own address and these addresses are used in the control program to identify each input and output device.

The operation of the PLC includes four phases which are repeated continuously for individual cycles of operation.

- Input Status Scan: During this period the sensor readings are mapped onto a data table called the input image table. This phase is carried out as a single step uninterrupted by other operations to provide a clear snap shot of the state of the process at a given instant.
- Program Execution: In this phase, the user control program is executed. The values in the input image table are examined, the required calculations and logic are performed and the results are stored in a data table called output image table.
- Output Status Scan: In this step the stored output values are sent to actuators and other field output devices.
- Housekeeping: In this step, several overhead functions such as diagnostic checks, service of peripheral devices, communications etc. are performed.

The time that a PLC takes to complete these four phases is called the scan time. The program scan time depends on the amount of memory needed by the control program and type of instructions used within it. The scan time can be usually calculated based on the instructions used in the control program. However, other factors such as use of remote I/O, execution monitoring of control program etc., can add additional scan time. PLCs are predominantly programmed using relay ladder logic. However, a variety of other languages such as instruction list, structured text, function blocks, sequential function charts etc. are now also used.

2.2.2 Process/Loop Control Systems

The heart of many process control operations is the loop controller [Earl92]. This is the device that does the actual control, keeping the process variable at the setpoint and

maintaining stability. Loop controllers are available as single loop and multi-function loop controllers. The loop controller may be broken down into three major functional areas: operator interface, control data processor and I/O interface. The operator interface typically includes visual display facilities for process variables and a keypad to enter setpoints and other control parameters. The control data processor is microprocessor based and includes system memory. The I/O interface consists of signal conditioning circuitry to convert the input and output to compatible signals for the control data processor and actuator control device, respectively. All loop controllers provide the capabilities for simple proportional-integral-derivative (PID) control without any requirement for programming. Other loop controller features that may be present include but are not limited to, auto-tuning capability, multiple PID algorithms, cascade control, dead time compensation, batch control recipes, diagnostics, internal clock, logic functions, math functions, adaptive control, feed forward control, fuzzy logic, etc.

2.2.3 Distributed Control Systems

The distributed Control System (DCS) [Wayn91] in its infancy, simply provided remote control of valves and other final control elements, based on setpoints from the operator and feedback from the process. Today, various control, interface, and communication functions are distributed among widely separated devices and a data highway carries information between them. Typically, several sophisticated multi-function process controllers form the network of distributed control operations with a centralized operations control room. This modular distributed architecture provides geographical and functional distribution. With geographical distribution, it is no longer necessary to run hundreds or thousands of separate wire pairs to link each point in a system with a central computer or control station. Instead, the system components can be located throughout a plant, all linked for plant wide communications via a data highway. To expand the system, the user simply connects new field devices to the data highway. Functional distribution means that control system tasks are assigned to individual devices. Thus, controllers in remote locations perform control functions independent of other devices in the system.

Similarly, both the centralized and remote operations consoles provide a real-time operator interface to observe process conditions and controller actions and to interact as necessary. DCS process controllers need not be programmed in the same sense as a PLC, since their 'programming' is more like choosing the right control function and configuring it. Their suppliers also provide a proprietary high level control language for programming if needed. Much DCS technology is proprietary in nature and used only in high end continuous process control applications due to the high cost of its implementation.

2.2.4 Computer Numerical Control Systems

Numerical Control technology has evolved from the "brittle" hard-wired analog control systems to flexible, reliable and performance intensive microcomputer based digital Computer Numerical Control (CNC) systems [Sten97]. The microcomputer acts both as a intelligent human interface and as a supervisory coordinating controller for spatially distributed, embedded microprocessor controlled drives. The distributed servomechanism drive controllers have the capability to communicate among themselves as well as with the supervisory microcomputer. CNC systems are programmed using standard part programming languages such as RS-274D. CNC systems provide a set of parameters that can be software configured to dictate the behavior of the system. They also offer a rich set of diagnostic messages and services. Distributed numerical control is a logical extension to CNC systems in that it replaces the earlier unreliable tape reader for part programs, with high speed storage and retrieval of programs through a factory control network. It provides facilities for control of program execution from remote computers, communication with other control systems for task coordination, collection of production statistics, etc.

2.2.5 Robot Control Systems

Most industrial robots use some form of PLC equipped with special motion control modules in their control systems [MC95]. The PLCs are augmented with coprocessors and software that lets them execute complex procedures according to simple

instructions in user programs. Such robots typically provide simple point to point or contouring motion. Complex and high performance robots use proprietary control systems technology to provide sophisticated features such as velocity control, force control, fuzzy logic control, vision system and mobility. Robot controllers are predominantly programmed using teach pendant and high level proprietary languages such as VAL and V+, but some provide facilities for using a systems language such as C. Robot controllers also provide capabilities for remote program storage and retrieval, remote program execution, communication and coordination with other controllers, and on-line status information from remote locations.

2.3 Factory Floor Communications

Many of the communication schemes for passing data among nodes (devices connected to network) on a factory automation network have been proprietary, and closely held by the companies that developed them. Interoperability between devices from different suppliers often requires gateways and special interfaces that can be inefficient, functionally limited and slow to be developed. Faced by demand for greater interoperability among equipment from various vendors, both at the control and plant information level, the industry focus has shifted to the development and adoption of standard protocols (sets of rules for formatting, encoding and transmitting data). In addition, the proliferation of increasingly smart devices at the field level and the growing functionality of these devices in control schemes have resulted in intense interest in the development of field level communication standard protocols.

All of these standards are based on the Open Systems Interconnection (OSI) reference model [ISO84] defined by the International Standards Organization (ISO). The OSI model defines seven hierarchical layers: physical, data-link, network, transport, session, presentation and application. At the bottom is the physical layer, which defines parameters such as bit rate, the method of encoding bits, electrical or optical characteristics of the communication channel and the manner in which stations are connected to the channel. The data-link layer organizes data into a sequence of bytes

known as a frame and passes it to the physical layer for transmission. The data-link layer also determines which station has a right to transmit on the network, a function called Media Access Control (MAC) that is critical to the performance of the network.

The network layer provides an end-to-end channel which could be made up of many point-to-point connections, or data links. The network layer also reroutes traffic to avoid congestion. The transport layer provides a reliable end-to-end transmission channel, regardless of how many links and subnetworks the data passes through. The session layer provides mechanisms for controlling dialogs between applications. In other words, it ensures that all participants in a dialogue encode their data in a common language. The presentation layer negotiates what is known as abstract syntax (the elements that make up a language vocabulary, such as integers, characters and records) and transfer syntax (the rules for representing these elements in 1s and 0s). At the very top is the application layer. This layer provides high-level services for data access.

A number of communication protocols oriented toward the control or field network level for industrial automation have been based on the OSI model. Notable among these are the Manufacturing Automation Protocol (MAP) and field level communication standards. The following sub-sections discuss these in detail.

2.3.1 Manufacturing Automation Protocol

MAP [Val92] was introduced as a means of interconnecting control devices such as process controllers, CNCs, PLCs and robots, and of providing a connection to higher-level plant systems. The MAP standard has been developed and is maintained by the Technical Committee for Industrial Automation, of the ISO. MAP is built upon the Institute of Electrical and Electronic Engineers (IEEE) 802.4 standard for physical and data link layers. This standard specifies coaxial or fiber connection in a bus or tree topology running at 5 or 10 Mbits/s at the physical layer. The data link layer MAP uses an approach called token passing to share access to the bus by multiple devices in a controlled manner. In a token passing architecture, the right to 'speak' on the network (the token) is circulated from device to device in a predetermined manner. As each device

receives the token, it can put traffic on the network for a predetermined maximum length of time before passing the token to the next device in the sequence. This gives every device on the network the chance to transmit with the maximum waiting time between transmissions for a given node being dependent on the number of nodes and amount of traffic on the network layers. Layers 3-6 of MAP make it possible for MAP networks to be interconnected with other networks and different data formats.

The Manufacturing Message Specification (MMS) [ISO90] is an internationally standardized messaging system for exchanging real-time data and supervisory control information between networked devices and/or computer applications. It defines the application layer (layer 7) protocol of MAP. The messaging services provided by MMS are generic enough to be appropriate for a wide variety of devices, applications, and industries. Whether the device is a PLC or a CNC or a robot, the MMS services and messages are identical. The MMS standard consists of six parts. Part 1 is the service specification and contains a definition of

- The Virtual Manufacturing Device.
- The services (or messages) exchanged between nodes on a network, and the attributes .
- Parameters associated with the Virtual Manufacturing Device and services.

Part 2 is the protocol specification and defines the rules of communication which includes

- The sequencing of messages across the network.
- The format (or encoding) of the messages.
- The interaction of the MMS layer with the other layers of the communications network.

The remaining parts explain how MMS can be used for a class of applications such as CNCs, Robot controllers, PLCs or Process Controllers. These companion standards model an application area in terms of objects which are then mapped onto MMS objects. To manipulate the application objects, one has to actually manipulate the corresponding MMS objects with the appropriate services. The key feature of MMS is the Virtual

Manufacturing Device (VMD) model. The VMD model specifies how MMS devices, also called servers, behave as viewed from an external MMS client application point of view. MMS allows any application or device to provide both client and server functions simultaneously.

MAP provides partial integration of plant devices, at the non-time critical layer of plant information gathering through a restrictive client-server model of communications. This and a host of other factors such as cost and complexity of implementation have led to the failure of MAP. However several smaller versions of MAP such as Mini-MAP, MAP/Enhanced Performance Architecture (MAP/EPA) and Factory Automation Instrumentation System (FAIS), have been developed and have received limited acceptance.

2.3.2 Field Level Networks

In the last two decades, much of the network integration work was focused at the plant information level, such as MAP. Recently, much of the network integration effort has shifted to the field level. This is due primarily to advances and cost reductions in microprocessor technology that have made possible smart field devices with digital communication capability. A field level network is a digital communications standard for measurement and control field devices. Field level networks are emerging at two functional levels: sensor bus networks used primarily as high speed communication networks for simple field devices, and low/high speed fieldbuses aimed at both the process control and/or discrete manufacturing industries [John95].

All of these networks have multidrop capability, allowing for the connection of numerous field devices to a single bus, rather than having each device wired directly to the control system. Consequently, the most obvious benefit of field networks is a reduction in the cost of field wiring and wiring maintenance. Additionally, digital communications makes it possible for devices to provide increased diagnostic information remotely and continuously. This in turn, reduces the amount of time and effort expended on field device maintenance. Further, since the smart field devices have some onboard intelligence, some

control functions can be further distributed downward to reside in the field devices themselves.

Sensor bus networks [McMa95] are aimed at replacing the point-to-point individual wiring currently used for photoelectric, proximity, pressure and other low cost sensors and switches commonly used in discrete manufacturing or packaging industry. Sensor bus networks require a four wire connection and are not capable of the same distances as fieldbuses. Two wires are used for communication and two wires provide power to the field device. Sensor bus networks are generally high speed with extremely simple message structures to enable millisecond response. Several sensor bus networks such as the Devicenet network, Smart Distributed Systems network (both based on Controller Area Network), Bitbus network, Highway Addressable Remote Transducer network, Actuator Sensor Interface network and LonWorks network, have been developed and have found varying degrees of acceptance.

There are three major low/high speed fieldbus protocols vying for international recognition [Chat92]. The first is IEC/SP50, which represents the joint efforts of International Electrotechnical Commission (IEC) and Instrument Society of America (ISA). It uses only three layers of OSI model, namely physical, data link and application layer. It provides peer-to-peer communications using twisted pair wire at several transmission speeds. The second is the Factory Instrumentation Protocol (FIP), a French national standard. It also uses only three layers, with the application layer services being a subset of MMS. It provides Master/Slave communications using twisted pair wire at several transmission speeds. The third is the Profibus or process fieldbus, a German national standard. It again uses only three layers with the application layer services being a subset of MMS. It provides master/slave or token passing communications using twisted pair wire at several transmission speeds. In addition, other fieldbus networks such as Controlnet network and Interoperable Systems Project fieldbus network have been developed and have found varying degrees of acceptance.

2.4 Open Architecture Control

Currently, most industrial control systems incorporate proprietary control technologies. Even though these proprietary technologies have been proven to be reliable and capable of meeting application needs, there are difficulties associated with using them. Examples of these difficulties include non-common interfaces, inability of equipment from different vendors to interoperate, and inability to extend and enhance the control system without replacing them. Hence, in recent years, industry and academia have shifted their focus towards the development of Open Architecture Control (OAC) systems. IEEE defines openness as [IEEE83]: "An open system provides capabilities that enable properly implemented applications to run on a variety of platforms from multiple vendors, interoperate with other systems, applications, and present a consistent style of interaction with the user".

The concept of OAC provides flexibility in terms of both hardware and software, and shifts the focus from hardware to software. By moving away from a hardware centric control to a software centric one, the OAC concept makes it possible to change the basic configuration of hardware at any time during the controller life cycle. This allows for incorporating advances in hardware without having to change the software interfaces. Further, the limitation of a restricted instruction set from a control equipment vendor, without the possibility for enhancement, is done away with in an OAC. Since the system is software based, the programming capabilities of a OAC is limited only by the Application Programming Interfaces (API) and the software libraries used. With the advent of mature OAC systems, the programming capabilities should approach that of general purpose computing systems.

An open architecture control system provides benefits such as reduced system costs, simplified integration of tasks, easier incorporation of diagnostic functions, better integration of user knowledge, and quick and easier reconfiguration of control systems with changing requirements. With the availability of open, modular control systems, the distinction between the various types of process/machine control systems and their

applications become blurred. The modularity and scalability of the OAC enables easy integration of particular functions for specific applications and hence reduces the need to have dedicated control systems.

As mentioned earlier, several initiatives are underway to develop an OAC. Notable among these are: the Open Modular Architecture Controller (OMAC), the Open Systems Architecture for Controls within Automation systems (OSACA), the Open System Environment for Controller (OSEC), the Enhanced Machine Controller (EMC), the University of Michigan Open Architecture Controller (UMOAC) and the Machine tool Open System Advanced Intelligent Controller (MOSAIC). The following sub-sections discuss these efforts in detail.

2.4.1 Open Modular Architecture Controller

The OMAC [OMAC94] initiative is a joint effort by the 'Big Three' automobile manufacturers in North America: Chrysler, Ford and General Motors, with a pilot project at the GM power train division. OMAC has the distinction of being the largest effort to develop an OAC to date. As an OAC, the openness and modularity of an OMAC are achieved mostly through software modules rather than hardware components. Fig. 2.2 illustrates the concept of modularity in OMAC using cooperating entities to perform the different controller functions. The scalability of the controller is achieved by adding, removing, or replacing control modules to the controller architecture. For example, modules for motion, sensing, and the network interface can be removed from the controller architecture to meet the requirements of a low cost control application. On the other hand, all these modules may be integrated to control a complicated, sensor adaptive controlled machining operation.

The modularity concept allows for interchangeability of controller modules i.e. replacement of a module with another that meets the same interface requirements even if the replacement module may not have identical, detailed internal functions. Instead of requiring replacement of an existing module, the model also allows for incremental functional improvements to each control module with technological advances and

changing requirements. The OMAC groups these modules into two sets: core modules and API modules.

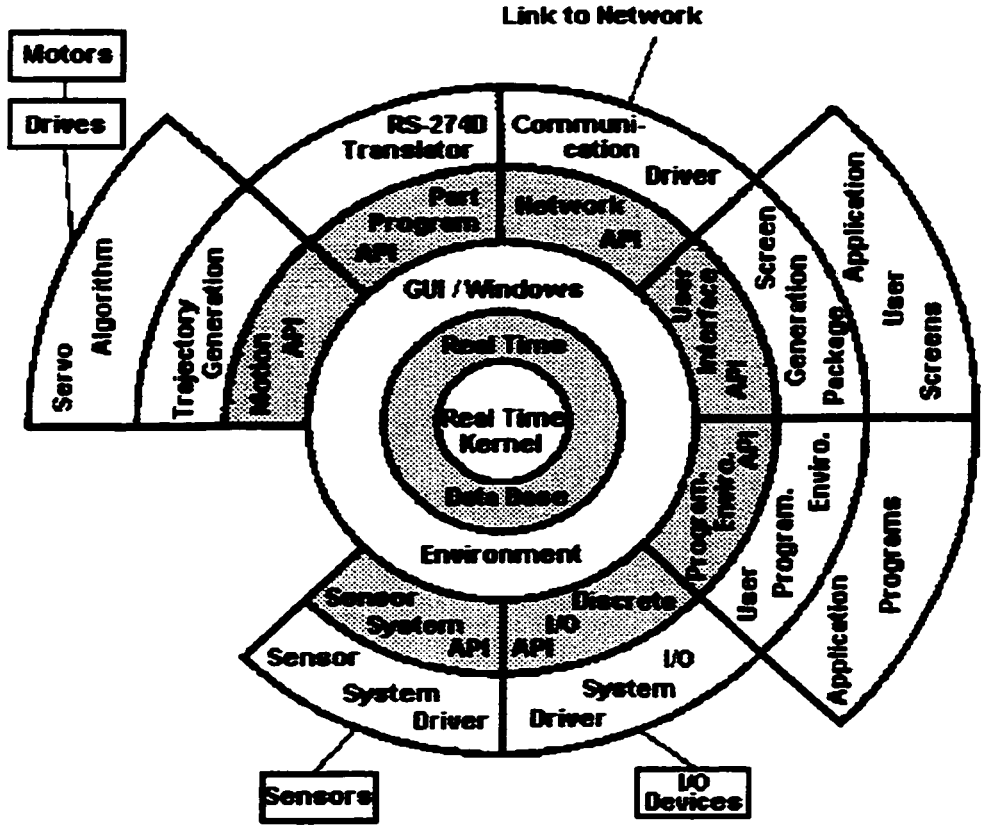


Figure 2.2: Open Modular Architecture Controller

OMAC identifies the real-time kernel, database, and the graphical user interface environment as being the core modules, and the successful resolution of issues in these areas as a prerequisite for its success. OMAC requires flexibility in terms of selecting the most appropriate operating system kernel for a particular application. In other words, a controller designed to satisfy applications with real-time requirements in the range of seconds may require an operating system kernel that is different from the one implemented in a controller that is used primarily in applications with millisecond real-time requirements. However, both controllers will have identical graphical interface environments to the users and this feature is considered critical to achieve scalability.

The API modules layer is considered to be the critical layer to achieve 'plug-and play' functionality and much work needs to be done in standardizing the interfaces. Through well defined and commonly accepted APIs, OMAC aims to integrate modules from various vendors into the controller infrastructure without extensive reprogramming, even though special efforts will still be required to integrate device specific software (e.g. device drivers).

As shown in Fig. 2.3, OMAC groups the functionality of control into eight controller elements. The infrastructure element consists of the hardware platform, real time operating systems, graphical user interface, and the underlying system level software that interacts with all other elements by sending and receiving information such as commands, status, and data. The information base element consists of a real time data base module and is responsible for storing, updating, and sharing system information and data that are needed for the machine or process to operate properly. The task coordination element functions as a coordinator of application tasks being executed by the controller. It ensures proper sequences of machine or process operations are scheduled and executed at the application level, by using the scheduling and coordination services of controller operating system.

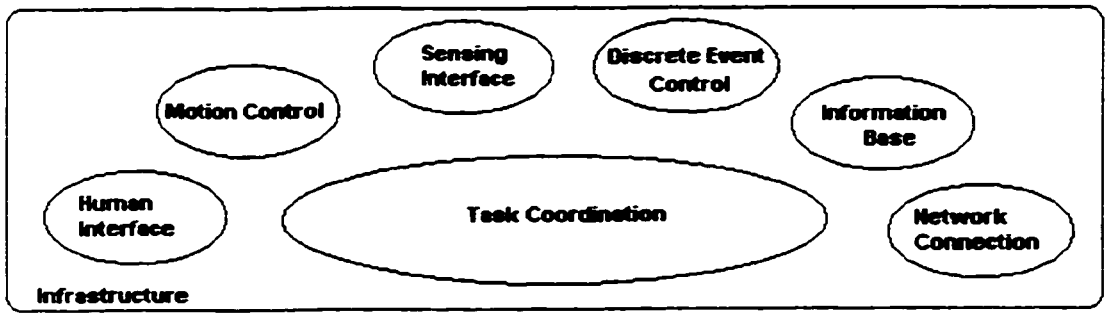


Figure 2.3: Elements of OMAC

The human interface element is used to input system parameters, program machine and process operations, operate the machine or process being controlled, monitor machine and process performance, display controller and process status, receive and display diagnostic information, etc. The motion control element provides the key functions for

path planning, trajectory generation, and servo loop trajectory tracking. The motion control functions may be executed by a dedicated motion control board in the controller or can be executed by the main CPU of the controller. However, the API between the infrastructure system software and the motion control element is identical regardless of the motion control hardware configuration.

The discrete event control element interacts with the external environment and implements an all software based PLC. It collects input information, executes the discrete logic, enables output devices, and also supplies I/O information to the real-time database for other controller elements to make proper decisions and take appropriate actions. The sensing interface element provides a means to gather information from complex sensing devices and systems, such as vision systems and force monitoring systems, that generally acquire and process a large amount of data. The network connection element provides facilities to upload and download programs and to transfer information about production statistics to the plant manufacturing information system when they are requested.

2.4.2 Open Systems Architecture for Controls within Automation Systems

OSACA [OSACA96] is a joint project by a consortium of European control systems and machine tool manufacturers and universities. The main goal of OSACA is to define a hardware independent reference architecture for controller equipment such as robot controllers, numerical controllers, logic controllers and cell controllers. OSACA has elaborated specifications primarily by defining a software reference architecture. This reference architecture enables the interchangeability and extendibility of comparable control specific application Architecture Objects (AO). It defines which AOs can be found in a control system, what tasks they perform and how they interact with each other.

As shown in Fig. 2.4, the architecture consists of two main sub divisions: the System Platform and Architecture Objects. The system platform consists of system hardware and system software. The system software contains core parts such as operating system, communication system, database and graphical user interface. It offers its services through a standardized API and is the only means of access from AO. This hides the

actual implementation of services, thus achieving hardware independence. Interoperability of AOs is achieved through a standardized communication system which not only allows interchange of data but also defines the protocols for interchange. Portability of AOs is guaranteed since the API is standard across various platforms. Scalability is achieved by adding, removing or modifying system hardware, software and/or AOs.

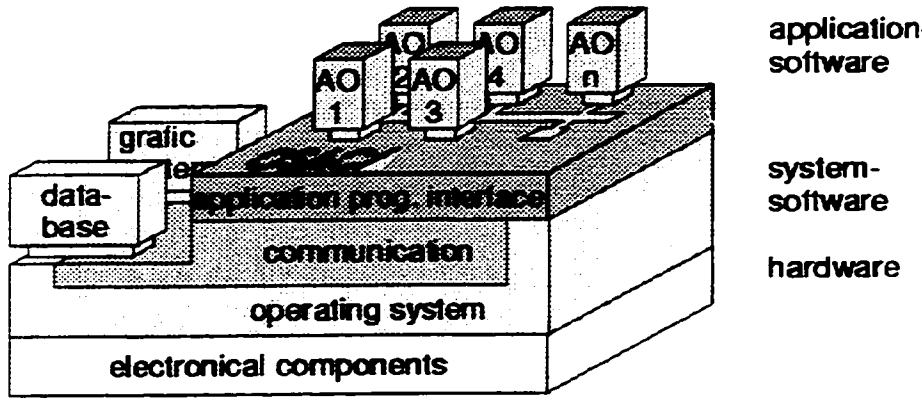


Figure 2.4: Open System Architecture for Controls within Automation Systems

The AOs are grouped into 5 areas also called subjects, according to their functions in control: Man Machine Control, Motion Control, Axis Control, Process Controls and Logic Controls. The man machine control represents the machine or part of it, to external entities, such as the operator and supervisory control system, and allows these entities to control the operation of the machine. Motion control enables the machine to produce relative motion of a given degree of freedom, by commanding axis controls. Axis control includes all the means necessary for activating the axis to execute movement commands within defined constraints. Process controls represent the auxiliary systems of the machine. Logic controls are responsible for discrete sensors and actuators of the machine.

OSACA has specified a vendor neutral communication system based on the ISO/OSI reference model. In this model, the internal control communication is via a uniform, message oriented communication interface. In order to fulfill the high real time demands on the controller, layers 1 to 4 were combined into the OSACA Message

Transport System and layers 5 to 7 to the OSACA Application Services System. The message transport system provides a hardware independent interface for transport of arbitrary messages between arbitrary objects both for local and distributed control. The application services system is responsible for connection management, encoding and decoding of messages, data format conversions and error correction within the communication system. To simplify the implementation and management of communication objects with the AOs, a Communication Object Manager is used. It provides an optional layer with standard routines and call back functions for creation and deletion of communication objects.

OSACA specifies that the real time controller operating system should be compliant with IEEE POSIX portable operating system standard and its real time extensions. It also specifies the requirements for process scheduling, real time aspects, parallel tasks and task distribution, within the operating system. OSACA specifies the architecture of a Configuration Manager that can be used for both static configuration during boot up and dynamic configuration during run time. Means are also specified for integrating databases.

2.4.3 Open System Environment for Controller

OSEC [OSEC95] is a joint effort by a consortium of Japanese machine tool and control equipment manufacturers to develop an open architecture controller for CNCs. OSEC has a restricted focus compared to OMAC or OSACA in the sense that it is only meant for CNCs. Further, OSEC architecture specifies a personal computer based open architecture Numerical Controller (NC). As shown in Fig. 2.5, the OSEC reference architectural model for CNC systems consists of the following parts: an operation planning part, a machining process control part, trajectory control part, an axis control part, a discrete event control part, a device control part and actual devices.

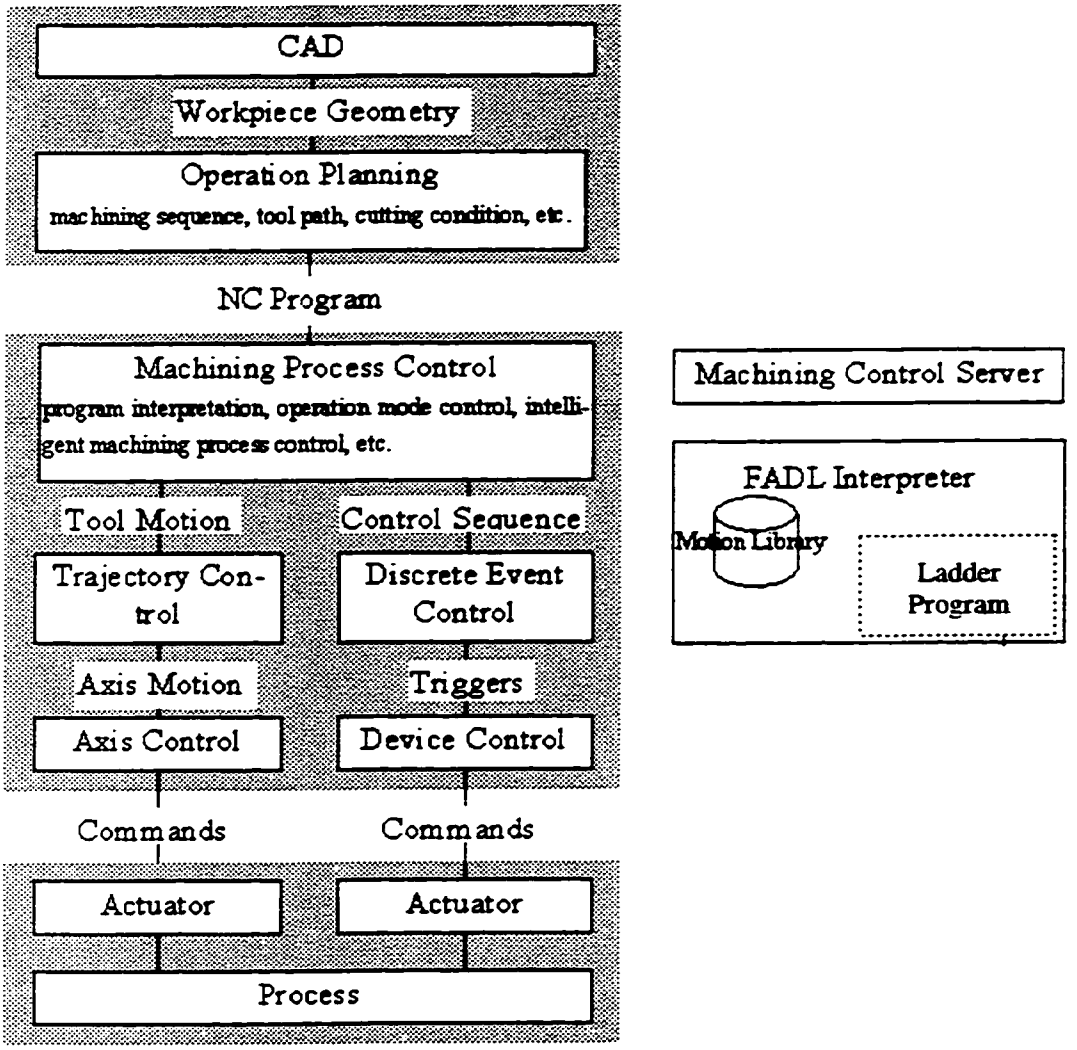


Figure 2.5: Open System Environment for Controller

The operation planning part includes a CAD/CAM system to generate geometric design, machining sequence and part program codes. The machining process control part includes program interpretation, supervisory operation control, intelligent machining process control, etc. The trajectory control is responsible for generating coordinated commands to axis control. The axis control is responsible for achieving desired axis motion. The discrete event control part executes logic control functions of the controller. The device control includes control of auxiliary devices that may be present in the machine tool. The actual devices include all forms of sensors and actuators.

The service and the protocol for connecting modules are designed to form autonomous agent systems that exchange messages in distributed network environments. To achieve a high degree of autonomy, OSEC defines a new data communication language called the Factory Automation equipment Description Language (FADL). FADL provides a rich set of services to interact with the real time controller and to command control sequence in a hardware independent manner.

OSEC also provides NC machining libraries to dynamically alter the control system performance. NC machining libraries are classified into 3 levels: machining description level, machining level and servo drive control level. These library functions are defined externally and linked dynamically during run time to command different types of system behavior and servo drives. The OSEC also specifies a service and message protocol for remote control of NC nodes based on the MMS standard. Service functions include management of the communication environment, uploading/downloading of program/data, reading and writing of control parameters, monitoring and signaling of alarms etc.

2.4.4 Enhanced Machine Controller

EMC [Proc93] is a joint effort by the Manufacturing Engineering Laboratory of the National Institute of Science and Technology (NIST) and the Department of Energy to develop an OAC as part of the Technologies Enabling Agile Manufacturing (TEAM) research. As shown in Fig. 2.6, the EMC architecture consists of a task sequencing component, a trajectory generation component, an operator interface component, a discrete I/O component, a servo control component and the sensors and actuators. The operator interface provides means to command and monitor the machine tool controller.

The task sequencing component is responsible for sequencing the high level commands to the controller. The trajectory generation component is responsible for generating coordinated motion commands for servo control. The servo control achieves the actual movement on a given degree of freedom. The discrete I/O component implements the logic sequencing operations of the controller. The EMC architecture is based on the Real Time Control System (RCS) model developed by NIST [Albus91,

Huang96]. The RCS model includes a software interface specification and library for real time controllers. It provides an exhaustive C++ API for control that is hardware independent and is portable across a variety of real time operating systems.

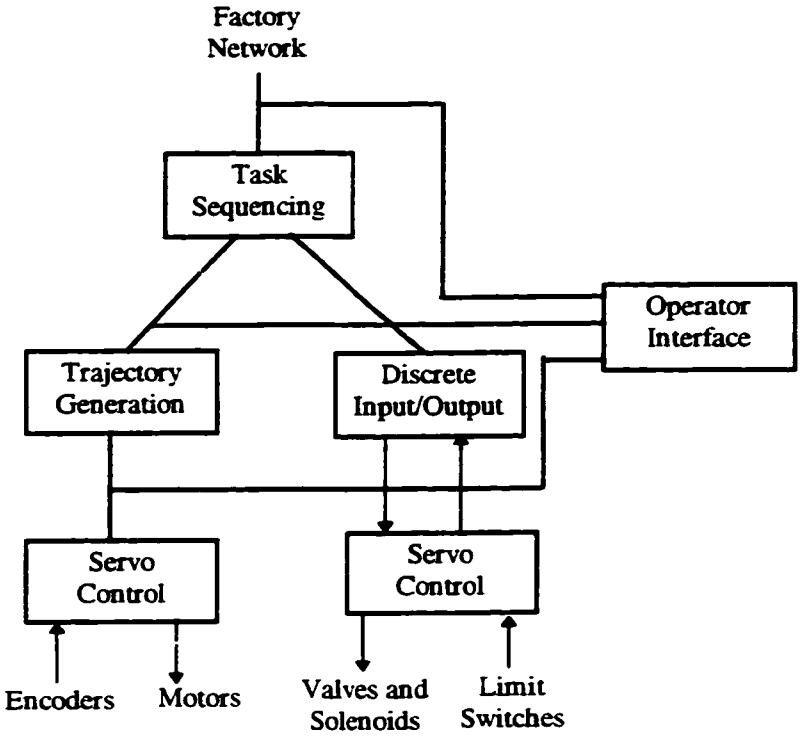


Figure 2.6: Enhanced Machine Controller Architecture

The interface specification is divided into two parts: commands that each module will perform, and status that each module will maintain. The specification also consists of messages 'into' each module, and world model data provided by each module. Supplementing the message specification is a model of data transfer based on the Neutral Manufacturing Language (NML). This model provides for "mailboxes" of data, with one or more readers and writers. Each module is modeled as a cyclic process, which reads its input command from its supervisor, reads the status of its subordinates (or sensors), and computes and sends outputs to its subordinates (or actuators).

2.4.5 University of Michigan Open Architecture Controller

UMOAC [Park95] is a research effort to build an open architecture real time controller for manufacturing systems. The base configuration of UMOAC is a distributed system in which processing nodes are connected through a real time link/bus. The architecture defines no particular hardware platform, but specifies that each processing node is based on an industry standard architecture and built with standard off-the-shelf components such as the VME bus. Three types of processing nodes are specified: operator interface node, real time computing node and real time control node.

The operator node is used for non-real-time tasks such as programming and non-real-time plant monitoring. The real time computing node deals with real time control and monitoring tasks such as real time data logging, diagnosis, scheduling and control. The real time control node performs fine grain real time tasks such as servo level control and data acquisition. The UMOAC has adopted the Controller Area Network (CAN) as the real time communication link between processing nodes and uses a Mixed Traffic Scheduling algorithm to support periodic, sporadic and non real time messages over the same CAN bus.

As shown in Fig. 2.7, the software architecture of UMOAC consists of three main layers: an application layer, an object management layer and a device driver layer. The application layer is composed of application programs, functional modules, and abstract machine models. Application programs are top-level software which include a user interface, programming and monitoring. The functional modules are hardware independent modular software components for control. The abstract machine model is a hardware independent representation of real machine hardware. This model also includes a detailed specification of data acquired during run time. The functional modules and abstract machine models are managed by an application integrator, with functional modules written for a specific application which can be reused for other applications.

The object management layer consists of the virtual device driver, system configurator, real time object manager and real time operating system. The virtual device

driver provides an hardware independent interface for I/O devices. The system configurator is responsible for mapping between hardware independent application layer software with real hardware and for providing a transparent view of system. For example, it maps the data to local device drivers for local I/O and to network driver for remote I/O.

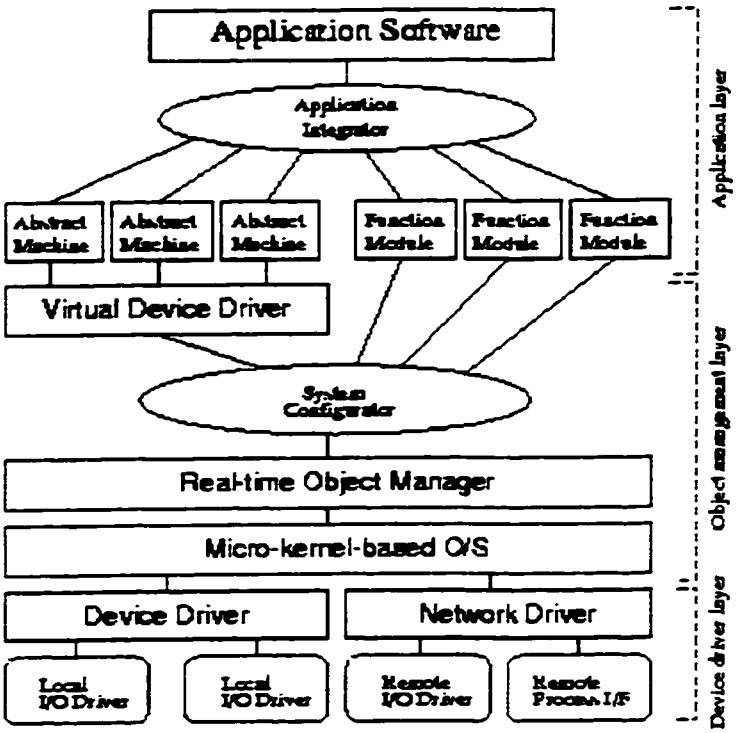


Figure 2.7: University of Michigan Open Architecture Controller

The real time object manager provides a object oriented wrapper for operating system services and include domain specific scheduling of tasks and resources. It also supports persistency, configuration definition and manages time critical data acquisition. The real time operating system is a micro kernel architecture commercial operating system, namely, the QNX. The device driver layer is the only hardware dependent part of the UMOAC architecture and provides both local and remote driver support.

2.4.6 Machine tool Open System Advanced Intelligent Controller

MOSIAC [Sarma95] is an OAC being developed by Integrated Manufacturing Laboratory, University of California, Berkley. The architecture is meant for CNC controllers and is currently part of the Integrated Manufacturing and Design Environment (IMADE) research. Fig. 2.8 shows the hierarchical level of MOSAIC within IMADE. The MOSAIC architecture consists of three major levels: trajectory planning level, real time interpolation level and servo level. The trajectory planning level generates coordinated motion commands for real time interpolation, which in turn achieves desired movement through servo control. The hardware of MOSAIC is based on standard off the shelf components such as processors and I/O cards, and is based on a VME system bus back plane.

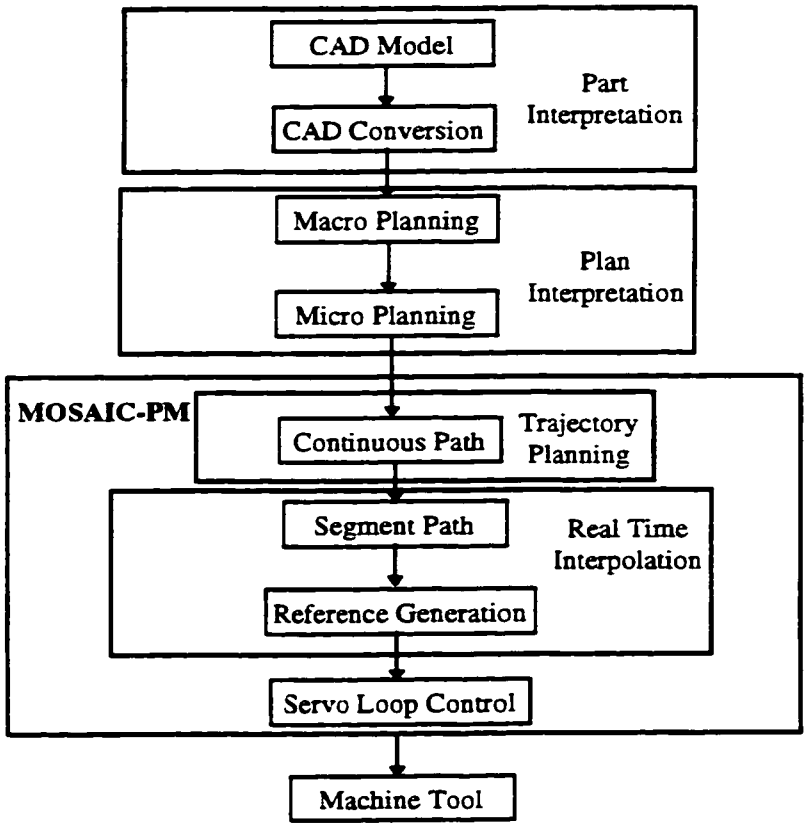


Figure 2.8: Machine tool Open System Advanced Intelligent Controller

The key to the MOSAIC implementation is a real time version of the UNIX operating system developed specifically for control purposes. This operating system provides capabilities such as multi-tasking, inter task communications, sockets based network communications and transparent integration multiple processor through the VME back plane. A library of software modules and an API in C is provided as callable building-blocks that run on the UNIX real-time operating system. This library of application program interfaces contains a variety of machine tool commands that enhance the machine's capabilities beyond simple RS 274 codes.

The API is categorized into primitives, functions and operations. Primitive-APIs are the simplest application programs that define high level behavior such as movement along a trajectory. Function-APIs are just more complex arrangements of primitives that define a parametrized machining macro. Operation-APIs are programs like function API but uses input from sensors in addition to the parameters passed. Additional complex machining algorithms such as adaptive control are implemented using these APIs.

2.4.7 Other Open Architecture Control Approaches

The Manufacturing Systems Integration Research Institute, University of Loughborough has developed an OAC as part of Integrated Machine Design and Control (IMDC) research [Harr96]. The IMDC is a software environment to build the machine and associated control system and to enable efficient modification as requirements change. Physically, the IMDC environment utilizes a network of one or more workstations or personal computers coupled to an embedded real time control architecture which resides on each target machine. The two main elements of the IMDC environment are a software tool set and a run time control architecture. The software tool set covers the life cycle of manufacturing machines and supports the creation of application software for the target control architecture. Underlying these elements is the IMDC system software written as set of C libraries which integrates and manages the user tool set and links it to the run time environment. The IMDC run time architecture provides the basis for an open, structured, device independent method for building machine control systems and is based on OS-9

real time operating system. Profibus is used as the main real time control network and enables control systems to be composed of intelligent devices, physically located at the locations in the machine where the control functionality is needed.

The Division of Production Engineering, Machine Design and Automation at Katholik University of Leuven has developed a Heterogeneous Distributed Real Time Architecture (HEDRA) [Deme95] for robot and machine tool control. HEDRA uses a heterogeneous multiprocessor hardware environment and a open, flexible software development and operational environment. HEDRA originates from an existing commercial real-time programming system called Virtuoso. The programming system contains a multiprocessor real time kernel as the operational part and an application development environment. The Virtuoso programming environment provides a software shield on top of complex multiprocessor hardware systems and provides a virtual uniprocessor environment for application development. A set of APIs are provided for various control tasks such as man machine interface, numerical control, robot control, axis control, process control and logic control.

The Manufacturing Engineering Laboratory of the University of British Columbia has developed an OAC for CNC machine tools [Yello96]. This controller is based on a PC and uses multiple processors in master/slave configuration for axes control. The controller uses the FORTH language and a set of programming libraries to define application level control tasks. The Advanced Mechatronics Laboratory of Carnegie Mellon University has developed an OAC approach for robotic systems [Stew94]. This approach uses CHIMERA, a real time multiprocessor operating system developed specifically for this purpose. A set of API modules based on port based objects is used by a graphical interface to synthesize application software assembly. The Aerospace Robotics Laboratory at Stanford University has developed ControlShell, a component based graphical real time software framework for open robotic control systems [Schn95]. The target run time environment uses VxWorks, a commercial real time operating system.

Automated Control Engine (ACE) [Eric96] is a commercial software that provides PC based event driven open architecture control. It uses a reusable library of control

component software to provide soft PLC functionality. In addition, several PC based open architecture CNC controllers are being developed commercially and Owen [Owen95] provides a comprehensive review of these efforts.

2.5 Summary

In this chapter, an overview of existing industrial control systems technology was presented. This technology is proprietary and suffers from number of drawbacks in the areas of interoperability, scalability and upgradability. This was followed by a discussion about the ongoing research initiatives in the area of open architecture control. These initiatives are shifting the focus of industrial control from being hardware oriented to software based one.

Chapter 3

Holonic Systems Control

3.1 Introduction

Traditional industrial control systems are unit level scan based regulatory control approaches or extensions of such, that use proprietary hardware and software. This limits the interoperability, programmability and upgradability of these systems. They provide network connectivity for programming, configuring and monitoring, and in some cases distributed sensing and actuation. However, this network connectivity is not the same as distributed control in a holonic sense. Open architecture control differs from traditional approaches primarily by using open controller infrastructure that provides software based control instead of hardware oriented control. This gives the controller much more flexibility in terms of off-the-shelf hardware and programming through the use of standard API's. However the fundamental nature of control remains unchanged and is still a unit level regulatory control approach.

The metamorphic control of holonic systems addresses a different type of distributed control problem that is not met by extant technology and approaches. Hence, in this chapter the differences and requirements of these control systems are discussed. This is followed by a review of research in the area of agent based control architectures for autonomous systems and a discussion on the applicability of underlying principles to holonic systems.

3.2 Metamorphic Control Requirements

A holonic system is inherently distributed and metamorphic, which has important implications for its control system. As shown in Fig. 3.1, a typical distributed control system of this nature makes use of a real-time network for communication among the spatially distributed controller nodes, and sensors and actuators. The centralized

application level control program that previously would have been executed on a single CPU is now distributed among the controller nodes. The distributed programs synchronize by communicating through messages under real time constraints to meet application requirements. The micro computer serves to program, configure and monitor the distributed control system. Metamorphic control requires numerous changes in form, substance and allocation to be accommodated within system life time.

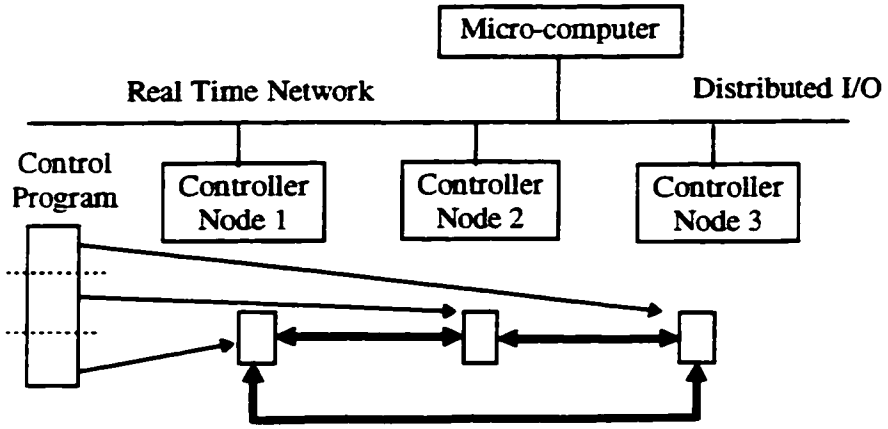


Figure 3.1: Holonic Distributed Control

A consistent specification model is needed whereby the application requirements can be partitioned and allocated across distributed controller nodes. Hardware independence and location transparency are fundamental requirements to entertain incremental changes, software reuse and reconfiguration. System level mechanisms are needed to meet the real time and communication requirements of distributed control applications. System level mechanisms are also needed to detect failure to meet such guarantees and take appropriate action. System level support is required to ensure reliability and fault tolerance.

Provision should be made at the individual controller node level for online addition, deletion or modification of specialized hardware and local I/O devices without requiring shutdown of the system for upgrade. Similarly, provisions should be made in the system software to recognize such changes during run time and map the application requirements accordingly. At a macro level, provisions are necessary to add, delete or

upgrade controller nodes, communication network and distributed I/O components dynamically during runtime without affecting existing distributed control applications.

Obviously, it is the responsibility of underlying system software to provide the necessary mechanisms for such a distributed system with real time performance, in contrast to a centralized system where no such specialized mechanisms are required. This necessitates a system level approach for developing such a control system and constitutes a fundamental difference from conventional control problem. For the same reason, conventional unit level approaches simply do not scale up and cannot be used for metamorphic control.

The complex nature of metamorphic control arises primarily from the unique combination and interaction of four system level requirements. These are the real time control, the distributed control, the event driven control and the intelligent control requirements. The following sub-sections discuss the nature of these requirements in detail.

3.2.1 Real Time Control

A real time system is defined as the one in which "the correctness of the system depends not only on the logical result of the computation but also on the time at which the results were produced" [Stan88]. Real time systems are characterized by the fact that severe consequences will result if logical as well as timing correctness of the system are not satisfied. Timing correctness requirements also arise because of the physical impact of controlling systems' activities upon its environment. For example, if the computer controlling a robot does not command it to stop or turn on time, the robot might collide with another object on the factory floor. In most of these systems, activities that have to occur in a timely fashion coexist with those that are not time critical. Ideally, the control computer must execute time critical tasks such that each task meets its deadline, and should execute non real time tasks so as to minimize their average response time.

Timing constraints for tasks can be arbitrarily complicated, but the most common timing constraints for tasks are either periodic or aperiodic. An aperiodic task has a

deadline by which it must finish or start, or it may have a constraint on both start and finish times. In the case of a periodic task, a period might mean once per period T or exactly T units apart. Low level application tasks, such as those that process information from sensors or those that activate elements in the environment, typically have stringent timing constraints dictated by the environment's physical characteristics and are mostly periodic. More complex types of timing constraints also occur. For example, spray painting a car on a moving conveyor must be started after time t_1 and completed before time t_2. Aperiodic requirements arise from dynamic events, such as a human operator pushing a button console or an object falling in front of a moving robot. In addition, time related requirements may also be specified in indirect terms. For example, a value may be attached to the completion of each task where the value may increase or decrease with time; or a value may be placed on the quality of an answer whereby an inexact but fast answer might be considered more valuable than a slow but accurate answer. In other situations, missing X deadlines may be tolerated, but missing X+1 deadlines cannot be tolerated.

Needless to say, lower level control systems such as servo positioning, cannot afford to miss timing constraints of critical tasks. Systems of this type where meeting the timing requirements is mandatory and missing deadlines will have catastrophic consequences are called Hard Real Time (HRT) systems. Resources needed for HRT tasks will have to be preallocated so that the tasks can execute without delay. However, many situations offer some leeway. For, example, if a control system estimates that the correct command to a robot cannot be generated on time, it may be appropriate to command the robot to stop without causing a different type of disaster. In this instance, the controller produces a lower quality result but on time. Systems of this type where meeting timing requirements is desirable, but a reduced quality of solution is acceptable and missing deadlines does not necessarily result in disaster are called Soft Real Time (SRT) systems.

In short, real time systems differ from Non Real Time (NRT) systems in that deadlines or other explicit timing constraints are attached to tasks and faults including timing faults may cause catastrophic consequences. This implies that real time systems

tightly interrelate correctness and performance. In addition to timing constraints, a real time system may have the following types of constraints and requirements: resource constraints such as access to I/O devices and data structures, precedence relationships among a related set of sub-tasks each requiring access to subset of resources, concurrency constraints related to consistency of a resource that is accessed simultaneously by several tasks, communication requirements among the cooperating tasks, and criticality depending on the functionality of an application.

A real time system needs to be both fast and predictable. High speed computing alone is not enough. Predictability has many meanings. In this context it is used to mean the ability to precisely determine a task's completion time with certainty. Predictability is dependent on the underlying real time operating system, the current state of the system and task's resource needs. Predictability also involves determining system performance under different levels of reliability. Reliability is a prerequisite for real time systems, since the constraints related to the system cannot be achieved if system components are not reliable. A real time system needs to be adaptive to changes in system state, system configuration and input task specifications. Adaptability is important because if a task's deadlines can be met only under restricted system state and configuration, reliability and performance will be compromised. On the other hand, if a system is adaptive one does not have to redefine the system or recompute resource and task allocation for every small change. A real time system's timing properties are very tightly related to the system hardware and the abstraction at which this binding occurs, in part, determines the adaptability and predictability.

3.2.2 Distributed Control

The holonic distributed control system can be considered to be comprised of a loosely coupled system of heterogeneous and asymmetric multiprocessor sub-systems. A loosely coupled system does not have a shared primary memory. Asymmetry means that every processor in the sub-system might not have access to same resources. Distribution implies a high degree of parallelism and asynchrony within the system and gives rise to

multiple failure domains among the components. Such systems differ dramatically from traditional ones by having

- concurrently executing tasks with different kinds and degrees of completion timeliness constraints
- close cooperative behavior among physically dispersed computing nodes having disjoint memory address spaces
- trans-node timeliness constraints and consistency properties
- inherent execution uncertainties, such as faults, dynamic dependencies, resource conflicts, overloads, and variable latencies.

Distributed control systems need to use real time network communication protocols that provide deterministic behavior for communicating components and a best effort service to deliver on deadline. Deterministic behavior requires protocols that result in bounded message communication delays where the bound is low compared to timing requirements. Real time performance and flexibility requires that both the producer-consumer (uni-directional) model and the client-server (bi-directional) model of communication be supported. Complicating factors in developing both types of services include the need to support high speed networks, the integration of high speed protocols with system software, I/O modules, and application modules, as well as the inclusion of dynamic reconfiguration and fault tolerant features.

Distributed control requires the maintenance of high degrees of trans-node consistency. Trans-node consistency refers to the properties which the system should maintain for correct behavior and operation. Sometimes the consistency objective is cooperation among nodes, whereas at other times the objective is non-interference. While all consistency properties can be expressed in terms of state, it is often convenient to differentiate among certain cases such as consistency of execution, data, computational groups, failure detection and recovery, and time. Consistency may be maintained explicitly or implicitly. Explicit maintenance of consistency involves the use of run-time facilities such as synchronizers and access protocols. Implicit consistency is an intentional or

unintentional consequence of the design and implementation of the system and applications.

In centralized systems with shared primary memory, explicit support is available for maintaining strong consistency of asynchronous concurrent execution and data accesses. The primary mechanism for this is mutual exclusion of memory, using synchronizers, such as locks and semaphores. In distributed systems, there is no shared primary memory, so explicit maintenance of trans-node consistency for execution and data accesses on multiple nodes cannot be implemented by mutual exclusion of memory, but has to be implemented with message passing communication. Hence deterministic and absolute inter-node consensus cannot be achieved due to inter-node communication and computation latencies, and multiple failure domains.

Further complications arise due to lack of an absolute and global system clock. In a distributed system every node maintains its own clock. Even if these clocks are initialized with the same time, physical clocks drift apart due to physical conditions such as temperature, so sooner or later, the individual nodes will have different time bases. This introduces the problem of clock synchronization. Due to the access and communication latencies of a real time network, though the clock synchronization may be bounded, absolute and infinitesimal resolution cannot be guaranteed.

Almost all extant commercial systems are based on the presumption that distributed systems consist only of decoupled local programs which have a very constrained range of simple roles and interaction behaviors i.e. they loosely interact in a two party client/server fashion. Hence, none of these systems support explicit maintenance of trans-node consistency in anything close to the sense used by all centralized systems i.e. application-specified and with arbitrary computation roles and interaction behaviors. For metamorphic control, it is a fundamental requirement that the system maintain trans-node consistency in a flexible and fault tolerant manner, to the extent supported by underlying hardware.

3.2.3 Event Driven Control

As shown in Fig. 3.2. traditional industrial control systems generally have scan based operation, in which the control tasks execute periodically in a non-preemptive manner. This is feasible since the system is centralized and both the local and remote I/O are accessed through image tables. In contrast, the control tasks in typical open architecture control systems are time triggered and execute in a preemptive manner. It should be noted that time triggered systems are a superset of scan based systems and may be extended to certain types of distributed systems. If the execution and communication requirements for a system are static and known a priori, and the execution and communication periods are static and known a priori, then such a system can be guaranteed deterministic performance through proper design and allocation in a time triggered system. However, when the system is dynamic both of these approaches will not suffice.

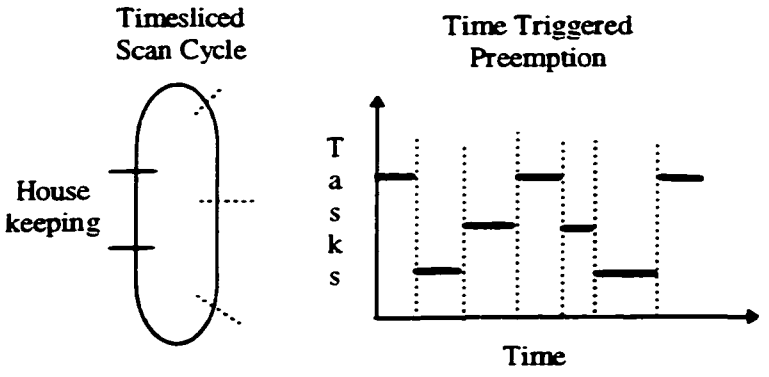


Figure 3.2: Scan Based and Time Triggered Systems

A metamorphic control system is by definition dynamic in nature. Dynamism is introduced into the system through numerous mechanisms inherent to metamorphic distributed control. Examples of such mechanisms include reconfiguration due to addition, deletion or modification of components and control tasks, fault tolerance, network access and communication latencies, and time variant task priorities and execution periods. Hence the dynamic behavior of the system can be adequately described only through the occurrence or non-occurrence of events. Naturally, it is necessary to use event driven

preemptive control for such a system. Therefore metamorphic control requires system level support for dynamic event driven control. As shown in Fig. 3.3, event driven control is a superset of time triggered control, since the expiry of a timer is an event and not vice versa.

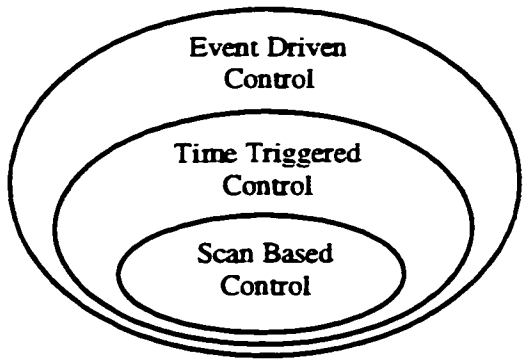


Figure 3.3: Control System Relationships

3.2.4 Intelligent Control

The control of holonic systems needs to incorporate intelligence in order to achieve the necessary basic characteristics for autonomy and cooperation. Intelligent control is multifaceted, since the mechanisms used will result in various forms and degrees of intelligence in the system. These include techniques such as distributed I/O based control, fault tolerant control, model based control and symbolic processing based control. Being a real time system, intelligent control needs to respond within deadline driven timing constraints. Hence metamorphic control should support mechanisms for highly reactive type of intelligent control. Higher notions of deliberative intelligence involve high variance reasoning and therefore are unpredictable. This leads to a trade off between available time versus quality of solution. Hence, higher forms of reasoning can act in parallel and may at times improve the system performance. Such interfacing requires uniform architecture through various levels and support at the real time system level for seamless integration.

3.3 Control Architectures for Autonomous Systems

As mentioned in the preceding section, holonic systems involve distributed autonomous control that is different from traditional approaches. Further, incorporation of intelligence into the control systems is an important requirement for holonic systems. Earlier research has established that agent based control paradigm is naturally amenable to distributed intelligent control of autonomous systems, as opposed to other paradigms of control. Hence, this section reviews agent based control architectures for autonomous systems. It should be noted that this dissertation uses the term agent to mean the control software components of a holon.

Agent based computing is an active area of extensive research in Distributed Artificial Intelligence. The notion of an agent is closely related to the concurrent actor model of computation. An actor is a self-contained concurrently executing object that has encapsulated internal state and responds to messages from other similar objects [Hewitt77]. However an agent is a more complex entity than an actor (and it may be added that the simplest form of agent would be an actor). Much of the research in agent based intelligent control has been done in the context of autonomous mobile robots and the scope of this review is limited to control architectures that were experimented with using actual robots.

Due to real time performance requirements, agent based control architectures tend to be substantially reactive in nature. Reactive agents do not possess an internal symbolic models of their environments, instead they react to raw sensory stimuli from their environment and according to present state [Ohare96]. Due to the absence of any world model, reactive agents are relatively simple and each typically defines a behavior. As they interact with other agents in basic ways, complex patterns of behavior and reasoning emerge from the dynamics of interactions, when these multi-agent systems are viewed globally. This emergent functionality is an important characteristic for reactive agent systems. Further, a reactive agent itself is comprised of a number of autonomously

executing concurrent objects that interact with each other through messages (i.e. like actors) [Maes91].

Subsumption [Brooks86] is the earliest reactive behavior based architecture that has been used experimentally and has since been refined considerably. Several other architectures have been derived from reactive behavior based architecture, including some that are hybrid in the sense that they also include deliberative components. The following sub-sections reviews these architectures.

3.3.1 Subsumption Architecture

The key aspects characterizing the behavior based robots of subsumption architecture are as follows [Brooks91a]: *Situatedness* characterizing robots located in the world and hence concerned not with abstract descriptions but with here and now and with the ability to directly influence the behavior of the system. *Embodiment* identifying robots as having bodies and experiencing the world directly and hence having immediate feedback on their sensations. *Intelligence* describable by an external observer and whose limitations are not intrinsic of the computational engine used, rather originated from situations in which the robot finds itself in the world, the signal transformation by the sensors and physical coupling of the robot with the world. *Emergence of intelligence* from the interaction with the environment and from interaction between the internal components.

Subsumption architecture [Brooks91b] is based on decomposing the control problem into levels of competence such that each layer is an activity producing sub-system. Fig. 3.4 shows one such activity decomposition for autonomous mobile robots. Each higher layer operates at an increasing level of competence and as an independent asynchronously executing module. The layers are composed of simple computational machines and communicate with each other through low bandwidth channels. However, the lower layers are oblivious to the presence of higher ones and are not dependent on proper functioning of higher layers. Such a decomposition allows incremental description of intelligence.

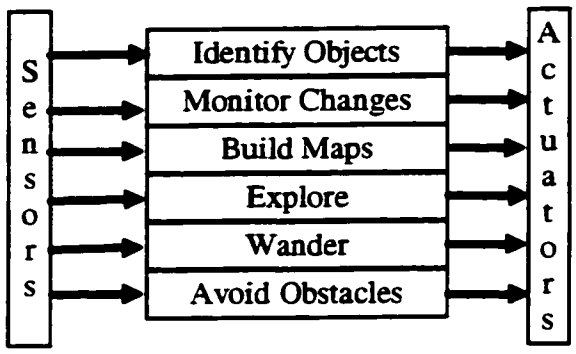


Figure 3.4: Activity Decomposition

As noted before there is no central representation of information of any kind. Rather, individual layers extract only those aspects of the world which they find relevant and every layer has its own purpose or goal. The lower layers are more reactive than higher ones and provide faster response to external stimuli. Because of the multiplicity of goals of various layers, conflicts exist. As shown in Fig. 3.5 the arbitration of these conflicts is resolved through the priority of the layers, i.e. the ability of higher layers to subsume the actions of lower layers, hence the name subsumption.

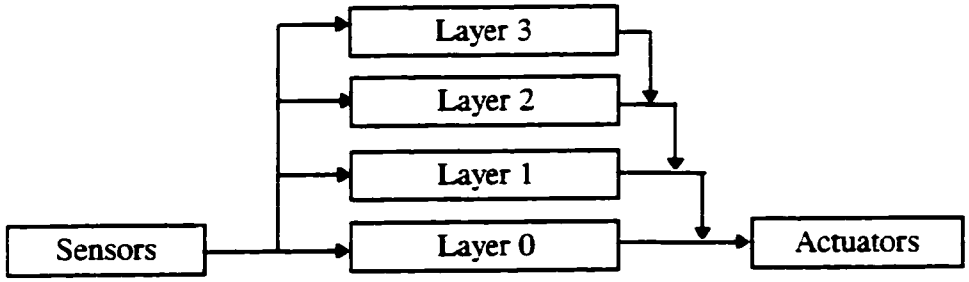


Figure 3.5: Subsumption Layers

Incremental enhancement in competence is achieved through the development and addition of successively higher layers in hierarchy and by guaranteeing that appropriate type of subsumption occur. This bottom up incremental development reduces the complexity associated with debugging the system. Additionally, changes in the environment have less chances of being reflected in all the layers, hence modification becomes easier. Since the competence is distributed among parallel executing layers, failure of a layer or unanticipated situations will not cause a total failure of the system. But it will result in functioning at a lesser competence level, thereby achieving robustness.

The layers of subsumption architecture are composed of network of behavioral agents that are "wired" with interconnections. The agents execute autonomously, concurrently and asynchronously by sending messages through wires. The agents act as abstraction barriers, one behavior agent cannot directly reach inside another. Agents of higher levels of competence subsume the actions of lower levels through suppression and inhibition. This is achieved through side tapping new wires onto existing wires. In the case of suppression, the side tapping occurs on input side of an agent. If a message arrives from a higher level on a new wire, this message is sent to the agent through the existing wire as though it has come through normal mechanism. Additionally, the input from the normal mechanism is suppressed for a short specified period of time. For inhibition, side tapping occurs on the output side of an agent. A message on the new wire inhibits messages being emitted on existing wire for a short specified period of time. However, unlike suppression, the new message does not masquerade as having come from original source.

A behavioral agent, in turn, is composed of a network of autonomously executing processes (concurrent objects) that communicate with each other. Each of these processes is an Augmented Finite State Machine (AFSM). Each AFSM has a set of registers and a set of timers, connected to a conventional finite state machine. The arrival of a message or expiration of a timer can trigger a change of state in the AFSM. AFSM states can wait on either wait on some event, conditionally dispatch to one of two other states based on combinational predicate or compute a combinational function and direct it to another AFSM. Much in the same way as agents, AFSMs can suppress or inhibit, input or output of other AFSMs.

The modularity of subsumption architecture facilitates several subsumption controllers to be operated in parallel, one each for various subsystems of controlled entity that cooperate to achieve total distributed control. The absence of a global shared memory due to the use of messaging paradigm facilitates physical distribution of control functions as well. Further, robust fault tolerant control is a natural characteristic of this architecture. Several robots based on the subsumption architecture have been successfully constructed and tested [Brooks91c].

3.3.2 Other Reactive Architectures

Several researchers have proposed extensions to the reactive behavior based control architecture which include some form of deliberative reasoning mechanism and goal driven behavior.

Goal driven behavior based control, derived from subsumption architecture, has been developed to integrate a distributed navigational map representation [Mataric92]. This approach does not attempt to make a distinction between reactive and planning systems. Instead, incrementally designed behaviors are used for collision avoidance, dynamic landmark detection, map construction and maintenance, and path planning. The topological representation uses primitives suited to the robots sensor and its navigation behavior. The map, unlike traditional centralized maps, can be characterized as a distributed collection of behaviors responding to the various landmarks, allowing constant time localization and linear time planning. The approach is qualitative and tolerant of sensor inaccuracies, unexpected obstacles and course changes.

The servo, subsumption and symbolic system hybrid layered architecture [Connell92] attempts to combine best features of conventional servo systems and signal processing with multi-agent reactive controllers and symbolic artificial intelligence. The partitioning of architecture into three layers is derived from a quantization of space and time. A centralized representation is introduced at the symbolic level, while the real time control aspects are delegated to subsumption and servo level. It bridges the gap between servo and subsumption layers by building situation recognizers and links subsumption and symbolic layers by introducing event detectors.

The intelligent machine architecture [Pack97] is a hybrid approach that brings together knowledge based and behavior based features. It uses a concurrent distributed network of agents that interact through a well defined set of relationships to realize the control system. Each relationship link encapsulates and manages a kind of action selection or arbitration mechanism that can be reused by agents in the system. At the highest level

are knowledge based skill agents, task agents and behavior agents, while the lower level is composed of reactive sensor and actuation agents.

The Reactive Action Package (RAP) [Firby94] is a task execution system that takes high level symbolic goals and refines them into a sequence of appropriate skill set. The RAP system uses a spatial planning module under its control to reason about navigational goals. The RAP system controls a set of reactive skill based continuous control modules to interact with environment. The skill set consists of action routines that interfaces with sensors and actuators. The skill set is selectively enabled by the RAP system according to the sequence required by high level system goals.

The design of the Atlantis [Gat91] architecture was based on the belief that competent behavior in a complex, dynamic environment demands different types of simultaneous activity. Quick reactivity is important for dynamism, but planning is necessary to deal with complexity. Atlantis supports three specialized layers, operating in parallel, to facilitate the required simultaneous activity. The control layer directly reads sensors and sends reactive commands to the actuators based on the readings. The stimulus-response mapping is given to it by the sequencing layer. The sequencing layer has a higher-level view of robots goals than the control layer. It tells the control layer below it when to start and stop actions. The deliberative layer responds to requests from the sequencing layer to perform deliberative computations.

In the Cooperative Intelligent Real time Control Architecture [Musl93], an AI subsystem reasons about task level problems that require powerful but unpredictable reasoning methods, while a separate real time subsystem uses its predictable performance characteristics to deal with control problems that require guaranteed response times. The key to this approach is to allow both subsystems to interact with each other without compromising their respective goals. To accomplish this, the architecture uses a scheduling module and a structured interface that allows the unconstrained AI subsystem to interact with the real time sub system asynchronously.

In the Distributed Architecture for Mobile Robots [Rose97], a set of task achieving reactive behaviors cooperatively determine robots control by expressing for and

against to possible course of actions in a distributed manner through voting. A centralized arbiter then performs a command fusion by selecting a course of action that is best suited to prioritized goals and constraints of the system. The architecture is designed with a belief that a centralized arbitration mechanism for distributed, independent decision making processes provides a coherent, rational, goal directed behavior, while preserving reactive real time responsiveness to the immediate physical environment.

A combined behavior based and cognitive control architecture [Doty95] supported by knowledge based perception has been developed. The principal components hybrid layered architecture decompose into independent, parallel and distributed functions. The behavior based component provides the basic instinctive competencies, while the cognitive part manipulates perceptual knowledge representations and a reasoning mechanism which performs higher machine intelligence functions such as planning. Cognitive control directly affects behavior through motivation inputs to behavior functions and through output behavior arbitration. The perceptual system offers a general framework for sensory knowledge generation, abstraction and integration, by fusing real time sensor data from multiple sensors.

The underlying principles of behavior based control architectures in reactive control of autonomous robots, hold promise for their applicability to generic holonic systems as well.

3.4 Summary

In this chapter, the differences and requirements for metamorphic control of holons as opposed to conventional approaches were described. This shows that the nature of research described in this dissertation addresses a fundamentally different type of control problem and that traditional approaches are inadequate to meet its requirements. This was followed by a review of research in the area of agent based control architectures for autonomous systems. The reactive behavior agent based control paradigm is ideally suited for distributed intelligent control of holonic systems.

Chapter 4

A Review of Real Time Systems

4.1 Introduction

The embedded computing technology of modern distributed industrial control systems is comprised of a number of diverse but inter-related components. Given such hardware, the efficient execution of a real time control application requires that programmers deal with issues that arise for high-performance, parallel and distributed application programs, such as efficient resource management, task and communication scheduling, load balancing, and programmed dynamic reconfiguration and recovery. These issues can be categorized into two levels: system level and application level. The system level determines real time capabilities of a controller, while the application level determines its sophistication.

The underlying operating system is the key to system level performance and the specification model is the key to application level sophistication. Hence the following sections discuss important results of earlier research in these two areas respectively.

4.2 Real Time Operating Systems

Existing commercial real-time operating systems are often stripped-down versions of general purpose time sharing operating systems such as UNIX. To reduce the run time overheads incurred by the kernel and to make the system fast, these systems typically are small in size, provide multitasking with fast context switch and external interrupt response, minimize worst case interrupt disable period, provide fixed or variable size partitions for memory management, and provide fast sequential file systems. To deal with timing requirements, they maintain a real time clock, provide a priority based scheduling mechanism, provide for special alarms and timeouts, and provide the ability for application tasks to invoke primitives to delay, pause and resume execution. Inter-task communication

and synchronization are achieved through standard primitives such as mailboxes, events, signals and semaphores. Most of these systems provide primitives that are compliant with IEEE POSIX 1003.1b [IEEE93] portable operating systems standard and its real time extensions. Examples of such systems include QNX [QNX93], OS-9 [Micro91], VxWorks [WRS94], pSOS [ISI93] and Chorus [Chorus96]. A complete list of existing commercial real time operating systems can be found in [RTE96].

Commercial operating systems are not designed for distributed control applications and hence become potential bottlenecks for non-trivial applications. The actual limitations are discussed elsewhere in this dissertation. However, there are research operating systems that are meant for distributed environment, even though under varying degrees of suitability for distributed control. The following sub-sections discuss the important issues and results for distributed real time operating systems, and review some of the extant distributed operating systems.

4.2.1 Scheduling

Scheduling involves allocating resources and time to tasks so that the system meets real time performance requirements. Consequently a scheduling algorithm is a set of rules that determine the task to be executed at a particular moment [Liu73]. For a given task set, if a feasible schedule exists, then the system is said to be schedulable and is called overloaded otherwise. It is often possible to find a polynomial time optimal algorithm for preemptive scheduling, while non-preemptive scheduling has been shown to be NP hard [Mok83]. Further, the delayed response due to non-preemptive scheduling makes it an unlikely candidate for complex real time systems. Research on real-time scheduling has experienced a major shift during the last few years, from static (off-line) to dynamic (on-line) scheduling. Thorough reviews of research in real-time scheduling appear in [Law83, Cheng88, Stan93, Kop93].

Early research work focused on relatively small-scale or static real-time systems, where task execution times and arrival rates can be estimated prior to task execution (i.e., data dependencies are limited), and where the resulting task schedules can be determined

off-line. The most commonly used static method is the Rate Monotonic (RM) scheduling algorithm[Liu73]. The rate monotonic algorithm assumes that all tasks are independent, periodic, preemptible, have constant execution time and that their deadlines coincide with the ends of their respective periods. The basic idea of the rate monotonic algorithm is to assign different and fixed priorities to tasks with different execution rates, with the highest priority being assigned to the highest frequency tasks, and the lowest priority to the lowest frequency task. At any time, the scheduler simply chooses to execute the highest priority task. Thus, by specifying the period and maximum computation time of each task, the behavior of the system can be categorized a priori. The rate monotonic algorithm is an optimal static algorithm, in that it can schedule a set of tasks if another static algorithm can do so.

One of the drawbacks of the rate monotonic algorithm is that its schedulable bound is less than 100%. A set of tasks is schedulable by the rate montonic algorithm if the following condition is met [Liu73].

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

where n is the number of tasks and U is the total CPU utilization by all tasks, given by

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \tag{2}$$

where C_i and P_i are the worst case execution time and period for task i respectively. From Eqn. (1), it can be seen that the schedulable bound decreases rapidly from 100% to 83 % for 2 tasks and to 78% for 3 tasks and so on. For a large value of n, the schedulable bound converges to $ln\ 2$ or 69 %. However these estimates are pessimistic and it has been shown that, for a uniformly distributed task set, the average case schedulable bound is about 88%. Further Eqn. (1) is only a sufficient condition and not a necessary one. In other words, a task set having an utilization factor greater than the values calculated per Eqn. (1) may still be schedulable, if each task meets its first deadline when all tasks are started at the same time [Liu73]. Another drawback of the rate monotonic scheduling algorithm is that it does not support dynamically changing execution times and periods that are crucial

for distributed and dynamic control systems. However, the rate monotonic scheduling algorithm remains the most popular since it can be used with commercial UNIX type real time operating systems.

Several modifications have been proposed to make the RM algorithm deal with aperiodic tasks. These include the background server, polling server, priority exchange server, deferrable server [Leho87] and sporadic server [Sprunt89]. A background server executes at a low priority, and makes use of any extra CPU cycles, without any guarantee that it ever executes. The polling server executes as a high priority periodic task, and every cycle checks if an event needs to be processed, if not, it goes to sleep until its next cycle and its reserved execution time for that cycle is lost, even if an aperiodic event arrives only a short time after. This results in poor aperiodic response time. In the priority exchange algorithm, a high priority periodic task is used to service aperiodic tasks. When an aperiodic tasks request exists, the server's time is used to execute this task, and when not, this time is exchanged with a lower priority periodic task. Instead of being lost, the run time of the server is then at a lower priority task and the schedulability of the aperiodic task is still maintained.

Like the priority exchange algorithm, the deferrable server policy uses a high priority periodic task to serve aperiodic tasks. Unlike the priority exchange algorithm, the server task in the deferrable server has a fixed priority, but may defer its computation time to a later point in the period if no aperiodic task is pending. If the server time is still not used by the end of the period, the time is discarded. On the other hand, if an aperiodic task arrives, the time is maintained and may be used to serve this task. This method of handling aperiodic tasks is easier to implement than the priority exchange algorithm since it does not have to handle exchanging of priorities. The cost of this simplification is a slightly decreased worst case periodic task scheduling bound. The sporadic server algorithm like the deferrable server uses a high priority periodic task for servicing aperiodic tasks. It differs from the latter in the way in which the computation time is replenished. Instead of being replenishing periodically, at fixed points in time, replenishment is determined by when requests are serviced. In the simplest approach, the replenishment occurs T units of

time after the budgeted time has been exhausted. The Sporadic server provides quick response to aperiodic tasks.

The problems associated with static scheduling algorithms have encouraged the use of online dynamic priority algorithms. The Earliest Deadline First scheduling algorithm can be used for both dynamic and static real-time scheduling [Liu73, Dert89]. As the name implies, this algorithm uses the deadline of a task as its priority. Since the task with the earliest deadline has the highest priority, the resulting priorities are naturally dynamic and the periods of tasks (represented by their deadlines) can be changed at any time. Further, a major advantage of this algorithm is that it has a schedulable bound of 100% for any task set. A major problem with this algorithm is that there is no way of guaranteeing which task will fail under transient overload. Transient overload arises in systems where the average CPU utilization is less than 100%, but worst case utilization is above 100%, leaving the possibility for one or more tasks failing.

A variant of earliest deadline first scheduling algorithm is Minimum Laxity (Slack) First scheduling [Mok83, Dert89], where a slack is assigned to each task in the system, and minimum slack tasks are executed first. Slack measures the amount of time remaining before a task's deadline will pass if the task uses its allotted maximum execution time. Essentially, slack is a measure of the flexibility available for scheduling a task. Like the earliest deadline first algorithm, this algorithm also has a schedulable bound of 100% for any task set. And in much the same way, it also suffers from the drawback of inability to guarantee the failing task(s) under transient overload. However, both these algorithms are optimal dynamic scheduling algorithms in the sense that they can schedule a set of tasks if other static or dynamic scheduling algorithms can do so.

The drawbacks of these two dynamic scheduling algorithms has resulted in another variant of deadline driven scheduling, called the Maximum Urgency First algorithm [Stew91], where each task is given an explicit description of urgency. This urgency is defined as a combination of two fixed priorities, and a dynamic priority which is inversely proportional to a task's slack. One of the fixed priorities, called task criticality, has precedence over the task's dynamic priority. The other fixed priority, called user priority,

has lower precedence than the task's dynamic priority. The idea is to use two user-specified notions of 'priority' to help on-line algorithms distinguish the importance of every task uniquely.

4.2.2 Synchronization

Synchronization is important in real-time systems for two reasons: tasks may experience unpredictable delays due to blocking on shared resources to which they require exclusive access, and solutions attained for synchronization may also help in constructing solutions for the multi-resource task scheduling important in several real-time applications. Theoretically, Mok [Mok83] showed that the addition of mutual exclusion requirements in realtime programs makes the general scheduling problem an NP-hard problem. In practice, a number of algorithms have been devised and evaluated.

For uniprocessor systems running periodic tasks, two recent protocols provide effective solutions to the scheduling problem with resource sharing. They are the kernelized monitor protocol [Mok83] and the priority ceiling protocol [Sha90]. In the kernelized monitor protocol, the earliest deadline first scheduling policy is used for task scheduling. All executions in critical sections are non-preemptible. However, schedulability analysis performed in this protocol requires the use of upper bounds on the execution times of all critical sections appearing in tasks. Since such upper bounds may be overly pessimistic, using the kernelized monitor protocol may result in low processor utilization.

The priority ceiling protocol is designed for systems where each task has a fixed priority and the rate monotonic scheduling algorithm is used. With this protocol, in the worst case, each task only has to wait for at most one lower priority task to finish in a critical section, and deadlocks cannot occur. Assuming that the longest possible waiting time is known for each task in the system, sufficient conditions for scheduling sets of periodic tasks can also be derived [Sha90]. However, the priority ceiling protocol cannot be directly used when priorities are dynamic, which is addressed in the protocol described in [Chen90a].

4.2.3 Communication

Deterministic real time communication is the back bone of a distributed real time system. To achieve this determinism, communication protocols must have bounded channel access delays and bounded message communication delays. The channel access delay is the interval between the instant at which a task issues a request for sending a message, and the instant at which the local communication interface actually transmits that message on the communication channel. The message communication delay is the interval between the instant at which a task requests the transmission of a message and the instant at which that message is successfully delivered to its destination [Panz93].

Two main strategies are used in sending real time messages: guarantee strategy and best effort strategy. In the guarantee strategy, an attempt is made to guarantee ahead of transmission time that the real time messages will meet their deadlines. The guarantee may be given during system design or operation, but the key feature being that once a message is accepted for transmission, it is guaranteed to meet its deadline. In the best effort strategy, the network will try to meet the message deadlines, but no guarantees are given. This strategy is used when there are insufficient network resources to meet all message deadlines and some of the applications can tolerate certain amount of message loss.

Hard real time messages fall into two categories: synchronous messages and asynchronous messages. Synchronous messages arrive periodically and must satisfy their deadlines or they are considered lost. Because synchronous messages are deterministic and all of their timing characteristics are known in advance, most of the research in this area has followed guarantee strategy. One popular network access arbitration based approach is rate monotonic scheduling with network wide global priority driven protocols [Plein92, Stros88, Stros89]. In this approach each message is assigned a priority and the access arbitration ensures that higher priority messages are sent before the lower ones. The rate monotonic algorithm attempts to assign priorities in such a way that synchronous message

deadlines are guaranteed. This approach is used over networks such as IEEE 802.5 token ring network and Controller Area Network.

Another popular transmission control based approach is timed token protocol [Kop94]. This is a token passing protocol in which the amount of time a node may hold time is bounded. This approach can be used with networks such as Fiber Distributed Data Interface and IEEE 802.4 token bus network. Another technique to guarantee synchronous message transmission is to use round robin scheduling among message streams. Networks using Time Division Multiple Access protocols can be used with this technique [Kuro88].

Asynchronous messages on the other hand, arrive randomly during runtime and must therefore be scheduled dynamically. This means that using the guarantee strategy is more complicated for asynchronous messages. If the maximum generation rate of asynchronous messages is known, then the messages may be guaranteed by allocating sufficient network bandwidth [Ram87]. The actual transmission may be accomplished by maintaining a periodic server for every asynchronous source or by dynamically estimating the feasibility for transmission based on network load.

The best effort strategy for asynchronous messages typically uses the Minimum Laxity First scheduling approach. The actual transmission is accomplished through priority driven protocols or window protocols. In priority driven protocols, dynamic global priorities are assigned to messages based on slack period and access arbitration ensures higher priority messages are transmitted first [Shin90]. In the window protocol, nodes in the network agree on a common interval or window. If the slack time of a message lies inside this window, it is considered for immediate transmission. If more than one message lies inside the window, the size of the window is reduced such that only one message remains. However, this process incurs substantial overhead [Zhao88, Znati91, Lim91].

4.2.4 Clock Synchronization

A global time base is an important requirement for the measurement of time instants at which events occur, intervals between events and to establish the causal order

of events in a distributed real time system. Distributed clock synchronization approaches include hardware solutions [Lala91], software solutions [Lamp82, Lamp85] and hybrid solutions [Kop89]. Despite the differences in these approaches, certain requirements have to be met by all of them [Kop87, Panz93]:

- The clock synchronization algorithm has to be capable of bounding, by a known constant, the maximum difference of the time values between the observation of the same event from any two different nodes of the system.
- The clock synchronization algorithm has to be capable of tolerating the possible fault of a local clock, or the loss of a clock synchronization message.
- The global time constructed by the synchronization algorithm has to be sufficiently accurate to allow a measurement of small time intervals at any time.
- The overall system performance is not to be degraded by the execution of the clock synchronization algorithm.

Clock synchronization algorithms can be classified into deterministic and probabilistic approaches. Deterministic approaches assume a maximum communication delay and use it to provide guarantees on maximum clock deviations [Lamp85, Schn87, Srik87]. If the delays experienced by n nodes receiving messages range between t_{min} to t_{max}, all of these deterministic algorithms can achieve a synchronization no closer than $(t_{max} - t_{min})(1 - 1/n)$ [Lund84]. The probabilistic algorithms capitalize on the observation that most messages incur communication delays shorter than the worst case delay. These algorithms provide a probabilistic guarantee with much smaller clock deviations [Crist89, Arvin89]. In other words, the guarantee may fail sometimes, but the failure probability is known or bounded.

4.2.5 Fault Tolerance

Given the safety critical nature of real time applications, real time systems must function in spite of failures. Fault tolerance requires error processing followed by fault treatment. Error processing typically takes one of two forms: error recovery or error compensation [Lapri88]. Error recovery replaces an erroneous state with an error free

state either from recovery points saved during past or by moving the system to a known state. Error compensation, on the other hand, involves providing enough redundancy in the system such that the system is able to provide acceptable level of services in spite of the failure of one or more of its components. Thus, this technique masks the faults in a system from its environment.

The complex interactive nature of a real time system with its environment means that error recovery techniques cannot be used for non-trivial applications. This is due to the fact that if the system were to be restored to a previous state or known state suddenly, certain states will not be undone or will be skipped in the process, thereby not producing the right kind of interaction between the control system and its environment. Hence the error compensation technique using spatial and time redundant architectures is widely used to mask faults in real time systems.

The number of redundant replicates needed depends on the types of failure that a system is designed to handle. Types of failures range from fail-stop/fail-silent components to components that experience Byzantine failures [Lamp82, Schl83, Ezhil86]. In fail-stop/fail-silent type failure the components are self checking and they either function correctly or do not produce an output at all. In Byzantine failure, some messaging components may malfunction thereby producing conflicting information about the system. Tolerating up to n fail-stop failures requires maintenance of $n+1$ replicates, while tolerating n Byzantine failures requires maintenance of $3n+1$ replicates.

The redundant components may be employed in active or passive manner. In active redundancy all redundant components provide service functions at all times, whether there is a failure or not. In passive redundancy one member serves requests another takes over when it fails. The responsiveness of active redundancy has resulted in its wide spread use in safety critical real time systems. The active redundancy is typically employed as one of N-Modular Redundancy (NMR) techniques [Siew84, Chen90b, Lo90]. The NMR techniques include N-modular redundancy with voting, N-modular redundancy with backup spares, N-modular redundancy with adaptive voting by non-faulty modules, N-modular redundancy with paired sift out and N-modular redundancy with comparison. Of

these the N-modular redundancy with comparison technique has the lowest complexity and best fault masking capability.

4.2.6 Distributed Real Time Operating Systems

Several researchers have developed distributed real time operating systems for applications such as distributed supervisory control and real time multimedia transactions.

ALPHA [Jen90] is a distributed operating system for large, complex, distributed real-time systems. ALPHA's kernel provides its clients with a coherent computer system on an underlying platform that may be composed of an indeterminate number of networked physical nodes. Its principle abstractions are objects, operation invocations, and threads. Objects are passive abstract data type in which there may be any number of concurrently executing threads. Objects exists on a single node and can be dynamically migrated among nodes. Threads are the units of schedulability and are fully preemptable. It is the locus of control point movement among objects via operation invocation which transparently and reliably spans physical nodes. ALPHA utilizes a transactional distributed computing model for trans-node concurrency control and integrity.

ARTS [Tok89] is a distributed real-time operating system for predictable, analyzable, and reliable distributed real-time computing environment. ARTS uses an object model layered on top of threads. Objects are implemented using the C++ with real-time extensions, called RTC++ [Ishi92]. Underlying ARTS objects are real-time threads which can be hard real-time or soft real-time. Real-time threads in ARTS use priority based rate monotonic scheduling method. The ARTS kernel implements an Integrated Time-Driven Scheduler (ITDS). The ITDS scheduler provides an interface between the scheduling policies and the rest of the operating system. An extended rate monotonic scheduling paradigm is used for communication scheduling. This allows the system to integrate message and processor scheduling with a uniform priority management policy.

Real-Time Mach (RT-Mach) [Tok90] is primarily an extension of ARTS distributed operating system and addresses the real-time aspects of threads, thread synchronization, inter-process communication, and other mechanisms to allow greater

predictability such as a tool-set for real-time program design and analysis. As in other operating systems RT-Mach augments the threads model with timing attributes and both periodic and aperiodic threads can have soft or hard deadlines. The ITDS scheduler of RT-Mach uses processor sets (which are collections of processors available to an application), with run queues specific to processor sets and provides five different scheduling policies (Rate Monotonic, Fixed Priority, Round Robin, Round Robin with Deferrable Server, and Round Robin with Sporadic Server) on each processor set, with primitives to get and set the scheduling policy. The IPC extensions in RT-Mach use priority-based queuing in message buffers. In addition, it provides primitives to propagate priorities from the sender of a message to the receiver.

Maruti [Gud90] is an object-based distributed operating system with encapsulation of services. Objects consist of two main parts: a control part which is an auxiliary data structure associated with every object, and a set of service access points. Timing information, maintained in the object, is dynamic and includes temporal relations among objects. Objects that reside in different sites needs agents as representatives on remote sites. Maruti is organized in three distinct levels: the kernel, the supervisor, and the application level. The kernel is the minimum set of servers needed at execution time including a dispatcher, a loader, a time server, a communication and a resource manipulator. Supervisor objects in Maruti prepare all future computations, ensuring their timely execution by pre-allocation of resources, whenever possible.

SPRING [Stan91] is a distributed real-time operating system for network of multiprocessor nodes. In SPRING, tasks are classified as critical tasks (hard real-time tasks), essential tasks (soft real time tasks), and nonessential tasks. The SPRING kernel supports both individual tasks and a collection of individual tasks (task groups) which have precedence constraints among themselves but share a single group deadline. Scheduling is accomplished through a four level scheduler mechanism that separates policy from mechanism. The design of the SPRING kernel provides for resource segmentation/partitioning, functional partitioning of processing requirements, selective resource pre-allocation, a priori guarantee for critical tasks, on-line guarantee for essential

tasks, integrated CPU scheduling and resource allocation, use of the scheduler in planning mode, separation of importance and timing constraints, end-to-end scheduling, and use of significant information about tasks at runtime, including timing, task importance, fault tolerance requirements and the ability to dynamically alter this information.

The HARTOS real-time operating system is meant for a distributed-memory architecture consisting of nodes connected in a hexagonal mesh [Shin91]. The operating system focuses on support for on-line scheduling, in part targeting applications in autonomous robot control and multi-media applications. HARTOS is built on top of commercial pSOS+ real time operating system. While the pSOS+ executive provides the low-level mechanisms for processor and memory management, HARTOS extends these for the multiple-node environment, and handles real-time communications. HARTOS specifically addresses the issue of fault-tolerant communications and local deadlines are imposed on each hop of a message rather than deadlines for end-to-end delivery.

The CTRON [Sak89] real-time operating system is a part of the TRON platform for industrial systems and is designed for network nodes consisting of different kinds of computers. To assure software portability, the operating system is subdivided into two functional sections. One section consists of functions that hide the processor architecture and provide common interfaces, and are not portable among nodes with different processor architectures. The other section offers portable functions that assume a common interface. The kernel interface is divided into four parts: a group for the common model, a group for the advanced real-time model, a group for the advanced complex function model, and a group for the advanced virtual memory model. The objects supported by the CTRON kernel are: tasks, synchronization, exceptions, timers, memory, interrupts, and black box.

MARS [Kop89] is a fault-tolerant distributed real-time operating system architecture for process control. The structure of the MARS operating system kernel differs significantly from that of the other systems because of the specific demands a distributed time driven system makes on its underlying operating system. MARS is designed to maintain a completely deterministic behavior even under peak-load conditions.

i.e. when all possible stimuli occur at their maximum allowed frequency. In MARS, all the activities are synchronous, and based on a globally synchronized clock. In particular, the only interrupts present in the system are clock interrupts, which marks both CPU and bus cycles, and this feature facilitates the system predictability. Task and communication scheduling is done off-line by a scheduler before runtime and stored in a runtime scheduling table, which is interpreted during runtime. Location transparency at the task level is achieved implicitly in the MARS system because tasks merely send and receive messages with a given semantics, but do not know the source and the destination of these messages.

It should be noted that these distributed real time operating systems were designed for specific application domains and hence are not suitable for event driven holonic systems.

4.3 Formal Specification Methods

The real time system specification method facilitates the design at application level through abstraction of functional and timing requirements. Hence it determines the ease of expressiveness, sophistication and usefulness of the system. Real time specification methods are usually based on techniques such as real time temporal logic [Alur89], timed petri nets [Ghez91], modechart [Jah88], timed I/O automata [Lynch89], timed process algebra [Baet91] and timed finite state machines [Raj93]. Of these, the finite state machine and its variations have found wider acceptance. Recent research in the area of specification methods has focused on object oriented approaches. These methods can be classified into two categories: concurrent object oriented languages and object oriented formalisms.

The programming language real time concurrent C [Geha91] is a superset of C/C++ that provides facilities to execute activities with deadline constraints, seek guarantees about timing constraints and perform alternative actions on failure of timing constraints or guarantees. RTC++ [Ishi92] extends C++ with temporally constrained objects called real time active objects and timing constraints may be specified both as part

of method declarations and as commands within methods. Flex [Kenn91] extends C++ with a language construct called control block and timing constraints can be specified inside a control block for predictable performance. DROL [Tak92] is a concurrent object oriented language that extends C++ with real time constructs. RealtimeTalk [Erik96] is an extension of SmallTalk language for real time applications, while Real time Java [Nil96] extends the Java language for time critical applications. RTSynchronizer [Ren95] extends the Actor programming language with language constructs for specifying timing relations.

The Object Oriented formalisms typically use a higher level visual specification technique for behavior and timing specification, and convert the requirements into standard languages such as C/C++ for specific run time environment. StateMate [Hare96] is a specification tool based on statecharts that is a variation of the finite state machine. ObjectTime [Gaud96] is a Real time Object Oriented Modeling (ROOM) tool that uses a variant of statecharts and the Actor model of computation. Onika [Gertz93] is a software composition system that uses port based objects for specification. ControlShell [Schn95] is an event driven finite state programming tool that uses objects called components and data flow among them to model the system.

However, industrial control systems need to use a specification model that is standard across various manufacturing applications. The IEC 1131-3 [IEC93] programmable controller languages standard offers a solution to unit level time triggered systems, but is not suitable for event driven distributed systems. On the other hand, the IEC 1499 [IEC97] Function Block architecture is an emerging standard for distributed industrial process measurement and control systems. It uses an explicit event driven model and also provides for data flow and finite state automata based control. The following sub-sections gives an overview of this object oriented formalism.

4.3.1 Reference Models

As shown in Fig. 4.1, an Industrial Process Measurement and Control System (IPMCS) is modeled as a collection of controller devices interconnected and communicating with each other by means of one or more communication networks that

may be organized in a hierarchical manner. The control functions performed by the IPCMS are modeled as applications. An application may reside completely in a single controller device or may be distributed among several devices. For instance, an application may consist of one or more control loops in which the input sampling is performed in one device, control processing is performed in another, and output conversion in a third.

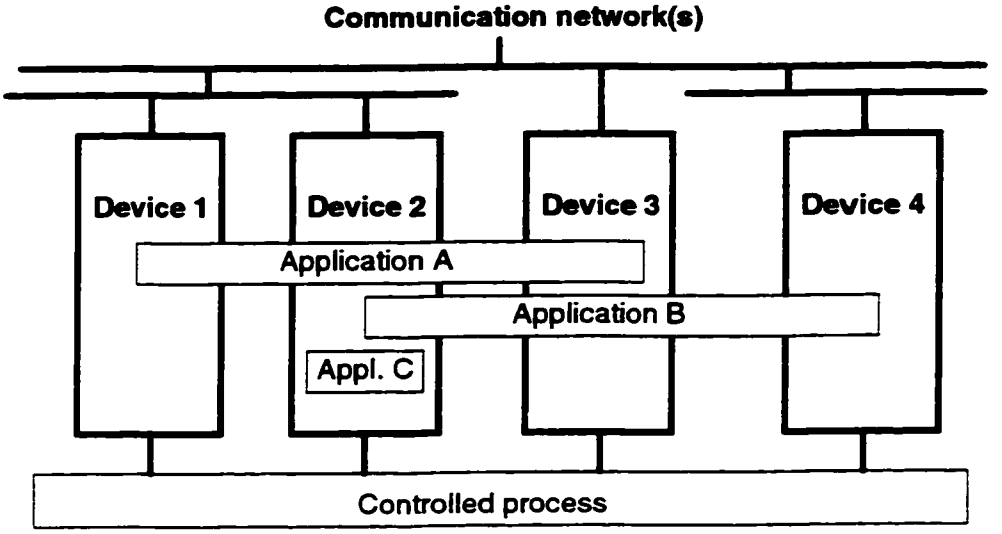


Figure 4.1: System Model

As illustrated in Fig. 4.2, a controller device contains at least one resource and at least one interface, viz., process interface or communication interface. A process interface provides mapping between the physical process (analog measurements, discrete I/O, etc.) and the resources. Information exchanged with the physical process is presented to the resource as process data or process events, or both. Communication interfaces provide mapping between resources and the information exchanged via communication networks. Services provided by communication interfaces include presentation of communicated information to the resource as communication data or communication events, or both, and additional services to support programming, configuration, diagnostics, etc.

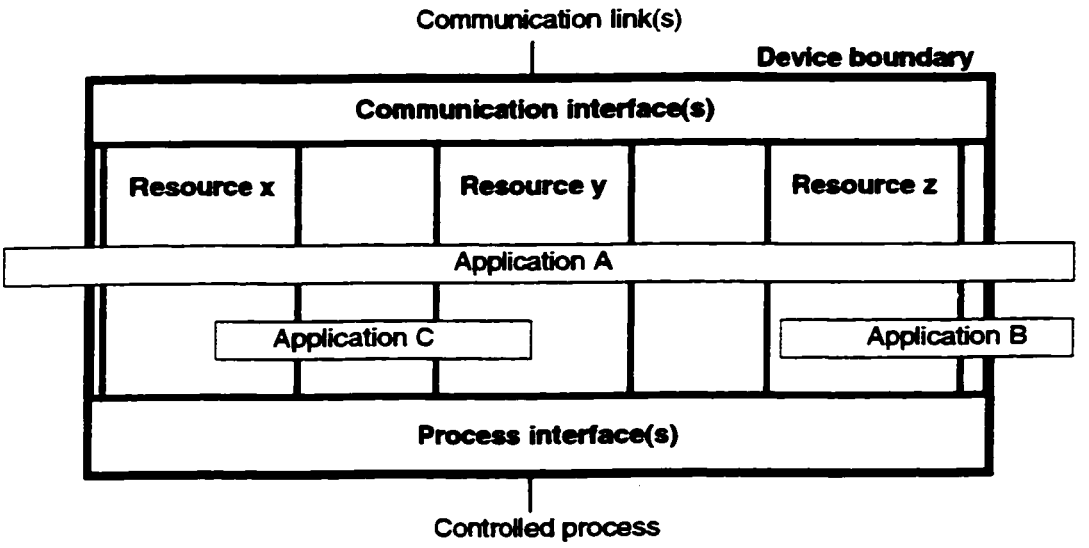


Figure 4.2: Device model

A resource is a logical subdivision within the software (and possibly hardware) structure of a device which has independent control of its operation. It may be created, configured, parameterized, started up, deleted, etc., without affecting other resources within a device. The functions of a resource are to accept data and/or events from the process and/or communication interfaces, process the data and/or events, and to return data and/or events to the process and/or communication interfaces, as required by the applications utilizing the resource. A resource provides the same functionality as an operating system (OS) and as illustrated in Fig. 4.3, a resource is modeled by the following:

- One or more local applications (or local parts of distributed applications) which are as independent as possible of the process and network worlds.
- A process mapping part whose function is to perform the mapping between process data and events and the variables and events used by function blocks.
- A communication mapping part whose function is to perform the mapping between communication data and events and the variables and events used by function blocks.

- A scheduling *function* which effects the execution of, and data transfer between, the function blocks in the applications, according to the timing and sequence requirements determined by the occurrence of events, the function block interconnections, the scheduling information such as task scheduling periods and priorities, and possible interactions with scheduling functions of other resources.

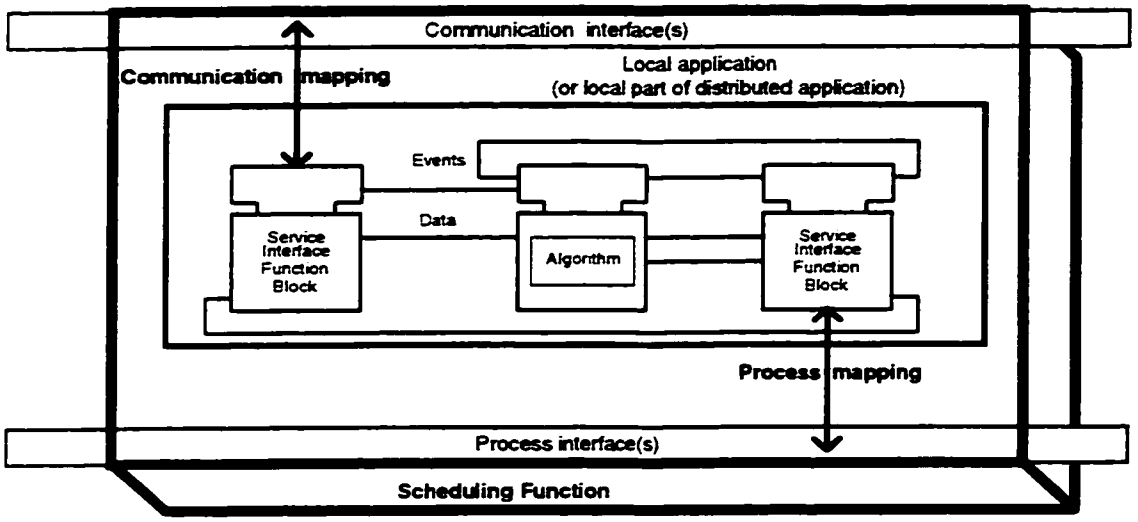


Figure 4.3: Resource model

An application consists of a network, in which nodes are function blocks and parameters and branches are data connections and event connections. An application may be distributed among several resources in the same or different devices. A resource uses the causal relationships specified by the application to determine the appropriate responses to events, which may include communication and process events. These responses may include scheduling and performance of operations, modification of variables, generation of additional events, and interactions with communication and process interfaces. As shown in Fig. 4.4, applications are defined by function block diagrams specifying event and data flow among function block instances. The event flow determines the scheduling and execution of the operations specified in each function block's algorithm(s) by the associated resource.

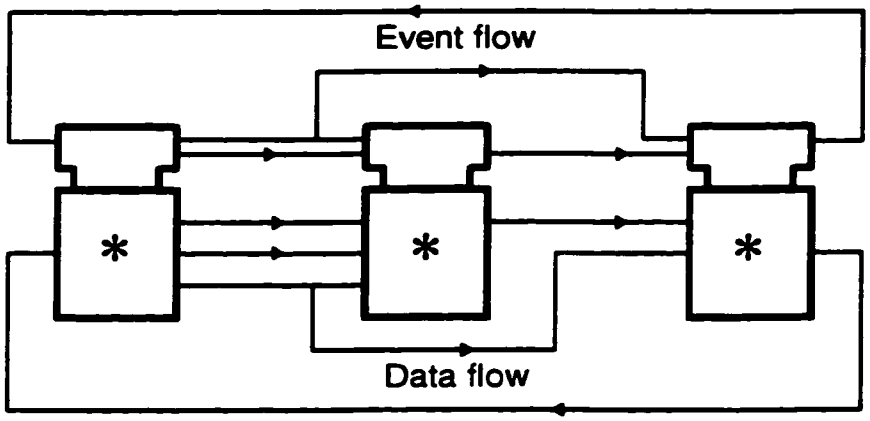


Figure 4.4: Application model

A function block instance is a functional unit of software comprising an individual, named copy of the data structure specified by a function block type, which persists from one invocation of the function block to the next. As shown in Fig. 4.5, a function block consists of a set of event inputs and outputs, a set of data inputs and outputs, a set of internal variables and state information, an execution control function and a set of algorithms. The execution control function, the algorithms, and the internal variables and state information which persist between invocations of algorithms, are invisible outside of the function block.

The execution of algorithms is invoked by the execution control portion of a function block instance in response to event inputs. This invocation takes the form of a request to the scheduling function of the associated resource to schedule the execution of the algorithm's operations. Upon completion of execution of an algorithm, the execution control generates zero or more event outputs as appropriate. Fig. 4.6 depicts the order of events and algorithm execution for the case in which a single event input, a single algorithm, and a single event output are associated.

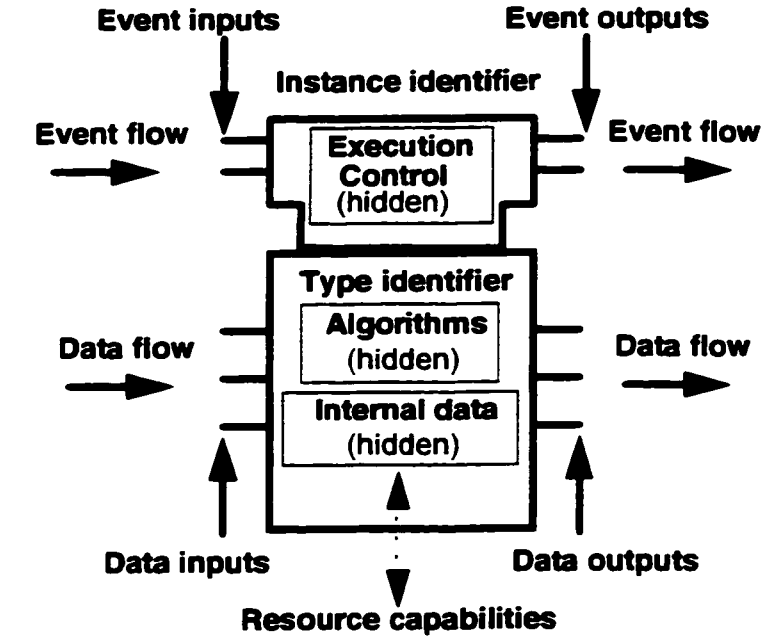


Figure 4.5: Function Block Model

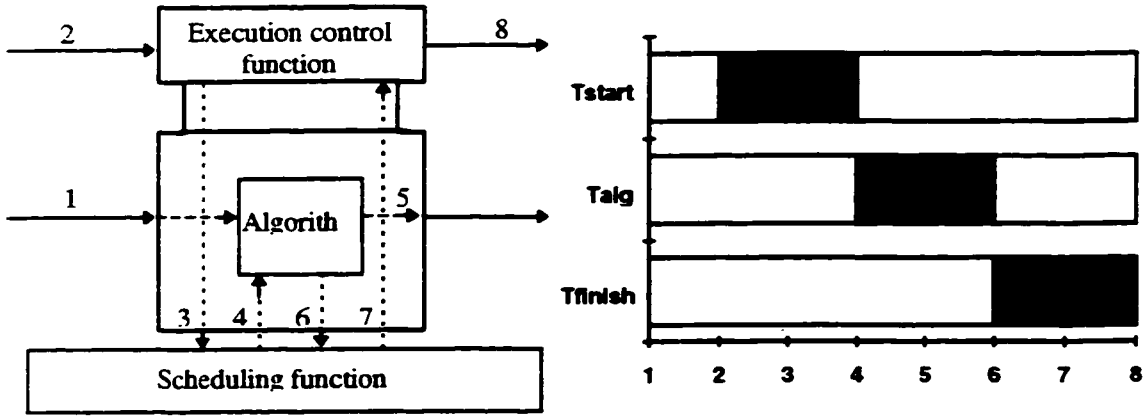


Figure 4.6: Execution Model and Timing

The times at which the sequence of low-level events occur in this case are as follows:

t_1: Relevant input variable values are made available.

t_2: The event at the event input occurs.

t_3: The execution control function notifies the resource scheduling function to schedule an algorithm for execution.

t_4: Algorithm execution begins.

t_5: The algorithm completes the establishment of values for the output variables.

t_6: The resource scheduling function is notified that algorithm execution has ended.

t_7: The scheduling function invokes the execution control function.

t_8: The execution control function signals an event output.

Fig. 3.7 shows the significant timing delays in this case which are of interest in application design. They are

$T_{start} = t_4 - t_2$ (time from event input to beginning of algorithm execution)

$T_{alg} = t_6 - t_4$ (algorithm execution time)

$T_{finish} = t_8 - t_6$ (time from end of algorithm execution to event output)

Due to these delays, various requirements exist for the synchronization of the values of input variables with the execution of algorithms, such as

- Assurance that the values of variables used by an algorithm remain stable during the execution of the algorithm.
- Assurance that the values of variables used by an algorithm correspond to the data present upon the occurrence of the event input which caused the scheduling of the algorithm for execution.
- Assurance that the values of variables used by all algorithms scheduled for execution in a function block correspond to the data present upon the occurrence of the event input which caused the scheduling of the first such algorithm for execution.

It is the responsibility of underlying resource to satisfy these requirements. The function blocks are classified into three types: Basic Function Blocks, Composite Function

Blocks and Service Interface Function Blocks. The following sub-sections give an overview of each type, respectively.

4.3.2 Basic Function Blocks

A basic function block utilizes an Execution Control Chart (ECC) to control the execution of its algorithms. As shown in Fig. 4.7, it also utilizes an explicit set of internal variables and state information, and algorithms. The ECC is a finite state machine similar to IEC 1131-3 [IEC93] Sequential Function Charts and is made of EC states, EC transitions and EC actions.

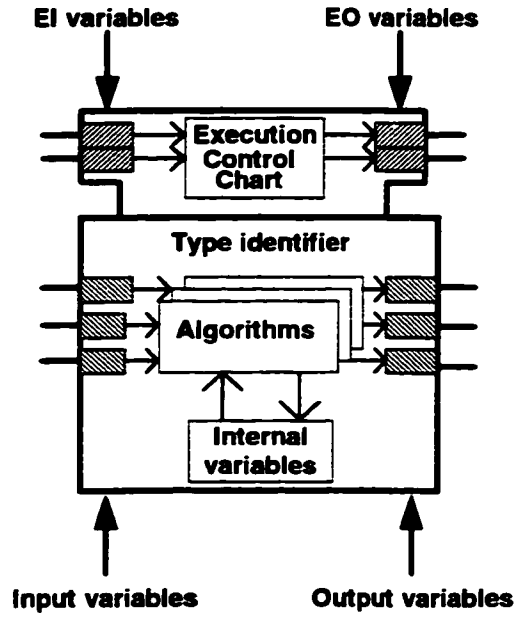


Figure 4.7: Basic Function Block

Fig. 4.8 shows a typical ECC of a basic function block. The initial EC state cannot have any EC actions associated with it. The occurrence of an event input causes the ECC to be invoked and the input variables to be mapped. The EC transitions use a Boolean combination of conditions that may be comprised of event inputs, input variables, output variables and internal variables. A "cleared" (i.e. triggered) EC transition causes a change in EC state and this leads to the execution of associated EC action. The EC action consists of zero or more algorithms associated with it. At completion of the algorithms.

the associated event outputs are set. The execution of algorithms may cause change of internal variables and state, resulting in other EC transitions being cleared. However, the evaluation of EC transitions is disabled till the completion of algorithms associated with an EC action. If no further EC transitions are cleared, the set event outputs and the associated output variables are issued.

As shown in Fig. 4.9, the resource maintains an event input state machine associated with every event input. Table 4.1 shows the actions associated with the transitions of this state machine. Transition *t0* occurs when there is a map input request without an event arrival at this state machine. The occurrence of transition *t2* means that same event has occurred before the request for first one could be completed. This may result in the loss of this event and the resources may provide mechanisms to detect and handle it in an implementation dependent manner.

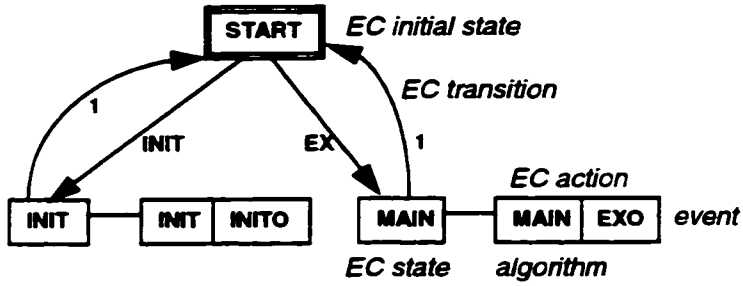


Figure 4.8: Typical Execution Control Chart

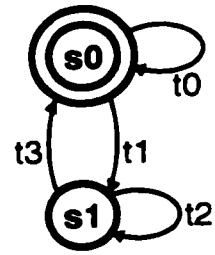


Figure 4.9: Event Input State Machine

Fig. 4.10 shows the ECC operation state machine and Table 4.2 shows the transitions associated with it. The ECC is invoked by the action associated with transition *t1* of event input state machine. This results in issuing a map input condition to all event input state machines of the function block, followed by evaluation of EC transistions.

Transitions *t3* and *t4* repeat till transition *t2* occurs. At this instance the output variables associated with set event outputs are sampled and the event outputs are issued. After which the event outputs are reset. Thus, the ECC controls the execution of a basic function block by cooperating with the associated resource.

Table 4.1 - Transitions of Event Input State Machine

Transition	Condition	Actions
t0	map input	none
t1	event arrives	ECC invocation request
t2	event arrives	implementation dependent
t3	map input	set EI variable and sample associated input variables

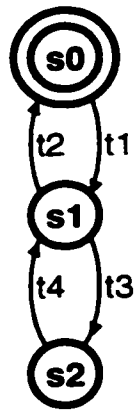


Figure 4.10: ECC Operation State Machine

Table 4.2 - Transitions of ECC Operation State Machine

Transition	Condition	actions
t1	invoke ECC	map inputs evaluate transitions
t2	no transition clears	issue events
t3	a transition clears	schedule algorithms
t4	algorithms complete	clear EI variables set EO variables evaluate transitions

4.3.3 Composite Function Blocks

The composite function block neither uses an ECC for execution nor does it possess explicit internal variables and algorithms. As shown in Fig. 4.11, it is composed of a network of interconnected component function blocks, events and variables. A component function block may be a basic, composite or service interface function block. The event and data connections specify the causality and sequencing of the component function block invocations. The occurrence of an event input at the composite function block causes the corresponding inputs to be sampled and invocation of all component function blocks. An event input of composite function block or an event output of a component function block can be connected to exactly one event input of component function block or one event output of composite function block. It needs to be explicitly split using a standard event processing function block in order to connect to multiple event inputs or outputs. Similarly, events will have to be merged with a suitable standard event processing function block.

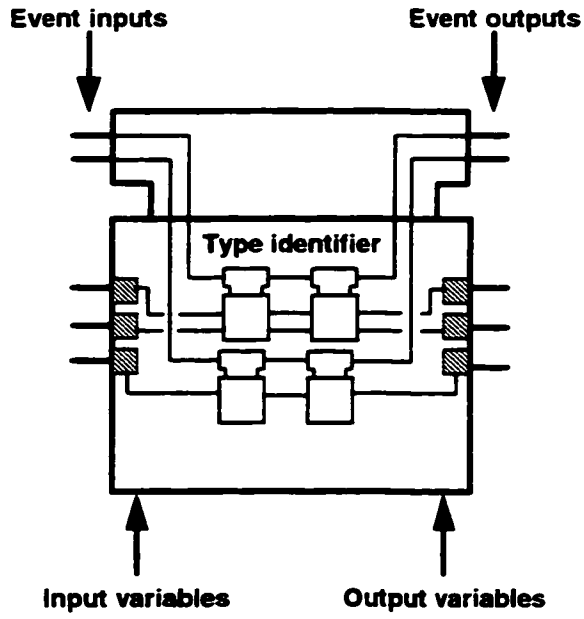


Figure 4.11: Composite Function Block

Unlike event variables, the input and output variables need not be explicitly split. However no two outputs may be merged (Boolean OR). The occurrence of an event output at a component function block causes the invocation of the next component function block in sequence or the issuance of the event output of the composite function block as required. It should be noted that event splitting may be used to invoke multiple component function blocks concurrently. Thus, the composite function block provides mechanisms for specifying complex behaviors.

4.3.4 Service Interface Function Blocks

A service interface function block is a function block which provides one or more services to an application, based on a mapping of service primitives to the function block's event inputs, event outputs, data inputs and data outputs. Basically, they provide a mechanism to map the services provided by the underlying resource, such as, process and communication mapping, configuration and management, etc. The external interfaces of service interface function block types have the same general appearance as basic function block types. However the semantics and behavior have special meanings.

Fig. 4.12 and Fig. 4.13 show the model of application initiated (requester) and resource initiated (responder) service interactions respectively. For the requester type function blocks the event inputs INIT and REQ are used to initialize and request service, while event outputs INITO and CNF are used to signal initialization completion and service confirmation respectively. The input variable QI interacts in a specific way with the input event depending on the logic level i.e. TRUE for beginning and FALSE for terminating the initialization or service. Similarly, the output variable QO is used to indicate success or failure of service.

The input variable PARAMS, which may be an aggregate, is used to pass the parameters associated with a service. The output variable STATUS is used to pass detailed information regarding service, typically on failure. The input and output variables SD_1, ..., SD_m and RD_1, ..., RD_n are used to pass application specific data for

service. The responder type function block uses event output IND and event input RSP to interact with an application.

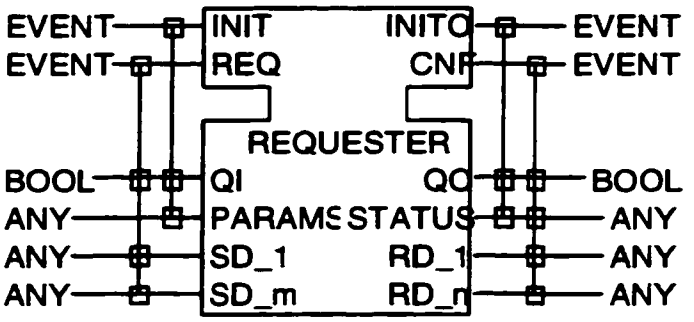


Figure 4.12: Application Initiated Interaction

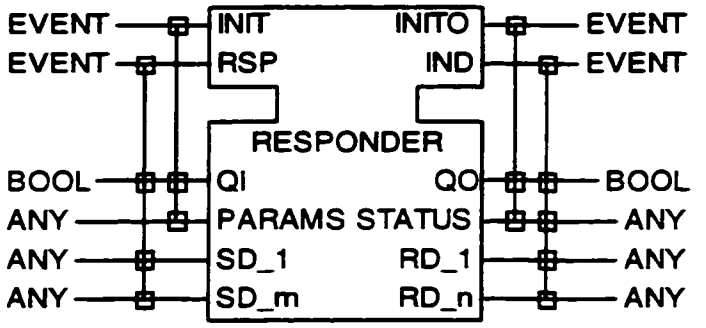


Figure 4.13: Resource Initiated Interaction

Communication function blocks are service interface function blocks that may be used for explicit trans node interactions in a distributed application. The publisher-subscriber and client-server model of communications are supported through unidirectional and bidirectional transaction communication function blocks respectively. Fig. 4.14 and Fig. 4.15 show the unidirectional requester and responder communication function blocks, while Fig. 4.16 and Fig. 4.17 show the bidirectional requester and responder, respectively.

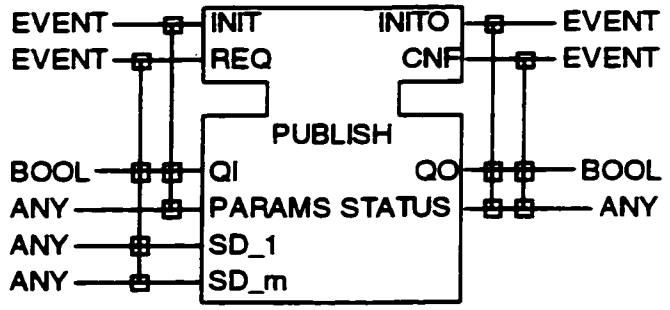


Figure 4.14: Unidirectional Requester

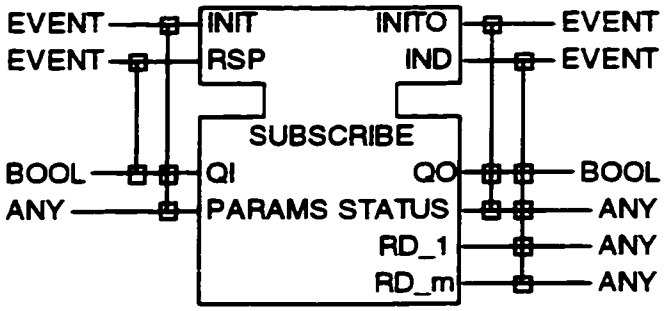


Figure 4.15: Unidirectional Responder

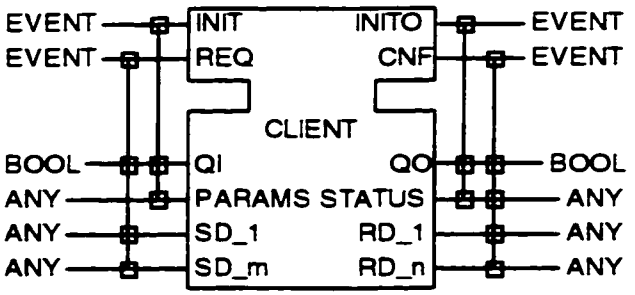


Figure 4.16: Bidirectional Requester

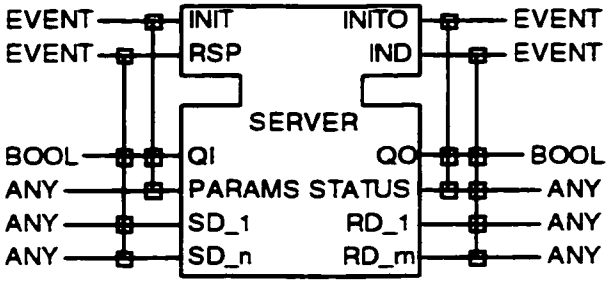


Figure 4.17: Bidirectional Responder

Manager function blocks are service interface function blocks that can be used to configure and manage applications. As shown in Fig. 4.18, manager CMD, OBJECT and RESULT variables instead of SD_m and RD_n variables. The standard defines a complete set of commands and valid syntax in each case, and the behavior of managed applications. The commands include create, delete, start, stop, kill, query, read and write services. Accordingly three classes for compliant systems are also defined.

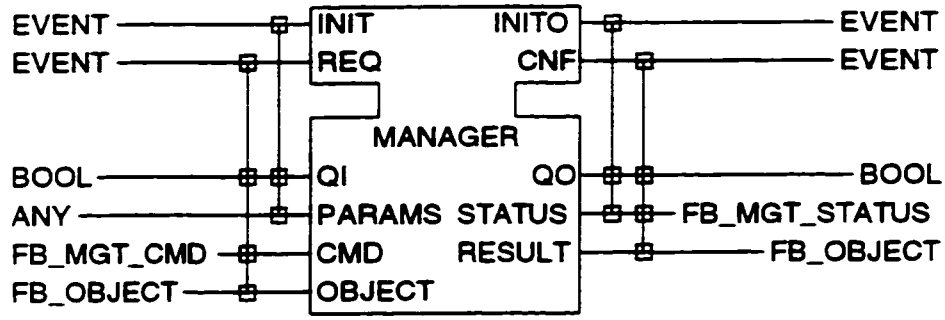


Figure 4.18: Manager Function Block

The versatility of IEC 1499 Function Block architecture makes it the most suitable real time application specification model for holonic systems.

4.4 Summary

In this chapter previous work related to the research presented in this dissertation were discussed. These included the important results in distributed real time systems theory such as scheduling, synchronization, communication, fault-tolerance and clock synchronization, and other research efforts aimed at developing a distributed real time operating system. Further, various approaches pertaining to formal specification of real time control applications were identified and the emerging IEC 1499 Function Block standard was reviewed.

Chapter 5

Metamorphic Control Architecture

5.1 Introduction

An adaptive/reconfigurable holonic system is able to evolve according to changing needs i.e. it enables easy re-design and allows addition, deletion and modification of system components during operation. A reference metamorphic control architecture that would provide unambiguous means for engineering such changes in form, structure and allocation needs to be defined. A reference architecture specifies a design method and style of building through abstraction of complex dynamic systems by simple models, interfaces and their integration. It describes the kinds of vital system components, their responsibilities, dependencies, possible interactions and constraints. By choosing suitable elements from this predefined set and using them appropriately, one can build a specific system with desired characteristics.

Evolutionary design of complex systems is characterized by the need to ensure incremental changes to the system will not introduce inconsistency and instability. Critical real-time systems pose an additional challenge as a high degree of assurance is required to ensure that a system continuously meets its timing requirements, even during periods of upgrade and transformation. Failure to do so can result in catastrophic damage to equipment and life. Hence the reference architecture and implementation must ensure incremental redesign and modifications will not lead to instability.

Accordingly, in this chapter a reference architecture for metamorphic control of holonic systems is described. A reference version of the architecture for metamorphic control is specified because it will be independent of any particular application and will capture all generic characteristics. In the following section, the architectural components of a metamorphic control system are identified and the concept of the distributed intelligent controller is introduced. This is followed by a physical controller architecture

that identifies the essential hardware components of the runtime environment. An Agent based uniform system and application software architecture is defined and the resulting functional architecture is presented. Finally, the critical issues in realizing this architecture and the components that provide means to engineer metamorphic control are discussed.

5.2 System Architecture

As shown in Fig. 5.1, the reference architecture for a metamorphic control system is comprised of four major physical components: microcomputer based system engineering hosts, primary factory control networks, distributed intelligent controllers and secondary Fieldbus networks. To some extent, these four components directly correspond to intelligent human interface block, inter holon interface block, process-machine control block and process-machine interface block, respectively. All the four components are softwired and can be dynamically reconfigured on the fly [Bala96].

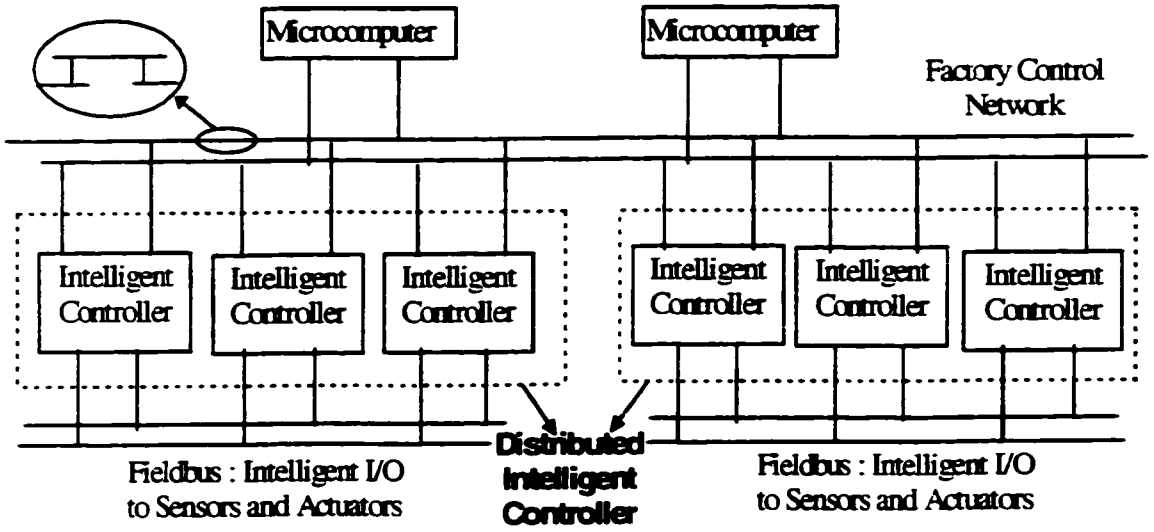


Figure 5.1: System Architecture

The microcomputers serve the purposes of system engineering interface functions, namely, programming, system configuration, status monitoring, data acquisition and possibly supervisory planning and control. The graphical user interfaces in these microcomputers are consistent throughout the system and all information about the system

can be accessed at any host subject to access privileges. Consequently, it is the responsibility of these interfaces to maintain consistent information and present a single logical view of the entire system. The primary communication network serves to control and coordinate all the distributed intelligent controllers within the factory control system. It also serves as the communication link between the controllers and the microcomputer hosts. The reference architecture provides redundant factory control networks for the purposes of reliability and fault-tolerance.

The distributed intelligent controller serves the purpose of controlling a holonic resource. A distributed intelligent controller is defined as a generic, open, autonomous and cooperative composite entity, comprised of multiple component modules called intelligent controllers, and capable of distributed real time intelligent control. It should noted that the physical structure of a distributed intelligent controller is scaleable according to application requirements and the characteristics of a distributed intelligent controller are due to its component intelligent controllers. The secondary communication network serves as the interface between one distributed intelligent controller and its distributed intelligent sensors and actuators and as a communication link among the constituent intelligent controllers. Here again, the reference architecture provides redundant Fieldbus networks for the purposes of reliability and fault-tolerance.

The reference system architecture may be modified in a number of ways during actual implementation. Fig. 5.2 shows two such modified and feasible system architectures. In first case, the primary factory control network is absent. A single secondary fieldbus network serves as inter-holon interface, intra-holon interface and as communication link with system engineering interface. In the second case, the primary factory control network and the secondary fieldbus networks are arranged in an arbitrary hierarchy through bridges and routers. Other possibilities include for instance, a distributed intelligent controller controlling more than one machine/process. Or, some of the intelligent controllers in a distributed intelligent controller may be connected only to the secondary network and thus do not have direct access to the primary network. Irrespective of such modifications, a core sub-set of features will be available across all

implementations.

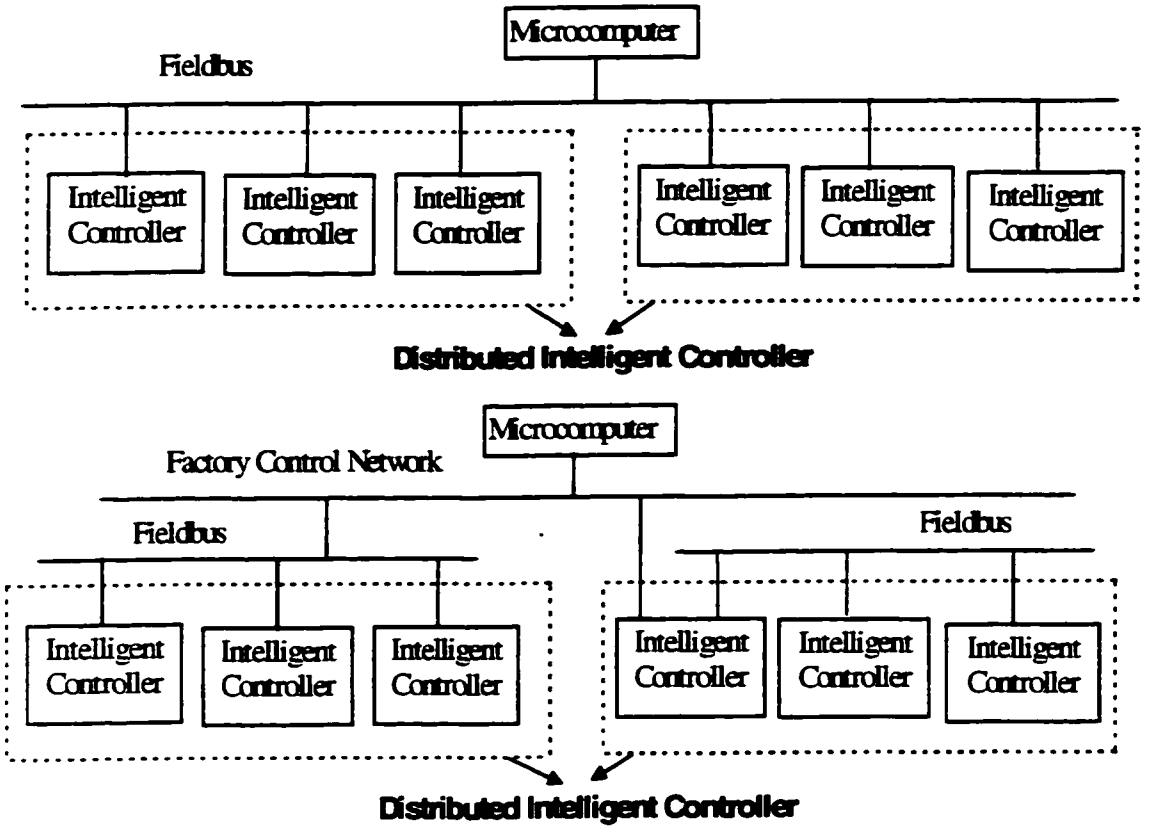


Figure 5.2: Feasible System Architectures

The following sub-section describes the nature of a distributed intelligent controller.

5.2.1 Distributed Intelligent Controller

As the name indicates, a distributed intelligent controller is by nature a real time system and involves prioritized execution of both hard and soft real time tasks concurrently. Distributed control requires that the diverse hardware and software components interoperate maintaining trans-node consistency and timeliness. Trans-node consistency requires that multiple nodes throughout the system. interpret and maintain the real world events and the associated data in a consistent manner. Timeliness requires prioritized and scheduled real time communications with predictable transmission

latencies. Intelligent control requires incorporation of various forms of intelligence that can enhance autonomy and cooperation.

As noted earlier, a distributed intelligent controller is comprised of multiple intelligent controllers and its characteristics are due to the component modules. The intelligent controller provides the physical platform, the run time environment and the mechanisms for a distributed intelligent controller. As a natural consequence, it is also an autonomous and cooperating entity. An intelligent controller in turn, can be conceptually visualized to be comprised of a number of logically and possibly physically separate autonomous and cooperating operational modules. Examples of such operational modules include the processor module, communication module, I/O module, system software module and application software module. The core functional characteristics of an intelligent controller can be conceptually generalized and categorized as follows:

- Autonomy
- Interoperability
- Reconfigurablity
- Fault Tolerance
- Real Time Functionality
- Intelligent System Interface

However, it should be noted that these characteristics acquire a different meaning depending on the functional level under consideration. Consequently, these characteristics also extend to the distributed intelligent controller as a whole. Further, as discussed earlier the intelligent behavior of a controller can best be characterized as being multi-faceted and dynamically adaptive. Hence, this dissertation considers the intelligent behavior of a distributed intelligent controller as dynamic emergent property due to the complex changing interactions among its components at multiple levels.

The following section describes a reference architecture for the physical modules of an intelligent controller.

5.3 Physical Architecture

As might be noticed from Fig. 5.1, the distributed intelligent controller is a loosely coupled system (no shared primary memory) of intelligent controllers. The reference physical architecture of an intelligent controller is modular in nature and may be comprised of a stand alone single board control computer or a tightly coupled architecture containing multiple processors. It should be noted that there are many possibilities for a multiprocessor architecture intelligent controller. However to be open, the architecture should be scaleable according to requirements and must utilize a defacto industry standard on processor independent, high performance, multi-master system bus architecture such as VME or PCI.

Fig. 5.3 shows two possible architectures for an intelligent controller: a uniprocessor architecture and a dual processor architecture. The minimal uniprocessor architecture intelligent controller is comprised of five modules, namely: a processor module, a programmable timer module, a main system memory module, a network communication module and a process instrumentation I/O module. The modules may be built as a single board control computer or may be built as 'plug-and-play' modules on a standard system bus architecture. This minimal stand alone control computer model doesn't provide a mechanism for graceful fault recovery by itself, but is theoretically enough for realizing an intelligent controller.

The processor module is comprised of a central processing unit, a memory management unit and a priority interrupt controller. The processor module ideally has nil overhead for the proper execution of other modules and they can interrupt the operation of processor module upon the occurrence of a significant situation. The data transfer between the modules occurs through the shared memory area or address space. The timer module is comprised of at least one counter/timer that could be programmed to provide real time interrupts for the processor module. The memory module comprises of the physical ROM and RAM. The network I/O module serves as the interface for the primary real time network communications among peers as described earlier.

The process I/O module provides the interface between the controller and the sensors and actuators. This module may be comprised of, a secondary network communication interface module if Fieldbus is used for instrumentation, or modular scaleable terminal I/O blocks if direct instrumentation is preferred, or both. The process I/O module provides facilities for both scan (synchronous) and event (asynchronous) based instrumentation. The modules and hence the controller operate in fail-safe mode i.e. either they function properly or they don't function at all, which simplifies fault diagnostics. Fault tolerance can be achieved through active redundant backup units and voting mechanisms.

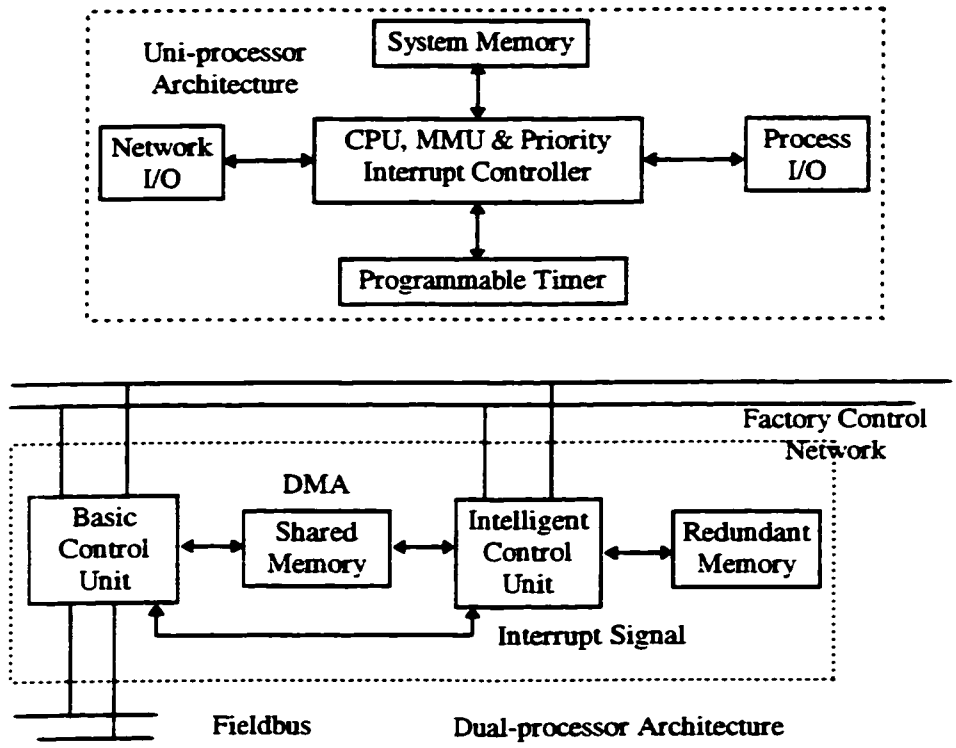


Figure 5.3: Feasible Physical Architectures

The reference dual-processor architecture intelligent controller would provide graceful fault recovery mechanisms. It consists of an intelligent control unit in a typical configuration with a basic control unit. The basic control unit includes a processor module, a timer module, a network I/O module and a process I/O module. The intelligent control unit includes a processor module, a timer module and a network I/O module, and

provides additional processing power. For example, the basic control unit might be used to execute hard real time tasks while the other may be used to implement soft real time intelligent control functions.

The control units and their modules may be built as a single board control computer or around an open industry standard system bus architecture. The primary memory is shared by both the control units through Direct Memory Access (DMA). The intelligent control unit maps the state of shared memory onto a redundant memory periodically according to a programmed cycle. The basic control unit and the intelligent control unit can communicate with each other either through the factory control network or through the shared memory. Either unit can interrupt operation of the other and can check functional status of other periodically, according to a programmed cycle.

The components in either unit are designed to operate in fail-safe mode. In case of a failure in one of the control units, the working unit can transfer the state information to another intelligent controller module with spare capacity for further action (passive roving redundancy). Active backup redundancies and voting mechanisms can also be supported. With this architecture, the distributed intelligent controller will be able to tolerate all single fails. For instance, a failure in basic control unit can be recovered with the help of intelligent control unit or failure of system memory can be recovered with redundant memory or a network failure can be compensated with redundant network on the fly.

Simultaneous fails can also be tolerated, provided that they are limited to one per functional unit. For example, simultaneous failures of a basic control unit in one intelligent controller and a intelligent control unit in another, coupled with a network failure can all be tolerated. It should be noted that to derive benefits from fault tolerant capabilities of distributed intelligent controller, the sensors and actuators must also be duplicated as appropriate. Further, the distributed control application software should also be suitably designed to take advantage of fault tolerant capabilities.

As said earlier, many other multiprocessor architecture intelligent controllers are feasible. For example, a uniprocessor controller closely coupled to a motion control module through the system bus or a uniprocessor controller closely coupled to a vision

processing and control module through the system bus or a uniprocessor controller coupled to a dedicated logic control module through the system bus. Irrespective of such implementation specific architectures, at least one processor will be open and would enable distributed intelligent control.

The following section describes an Agent based uniform software architecture for system and application levels of distributed intelligent controller.

5.4 Software Architecture

The reference software architecture of a distributed intelligent controller is specified as a heterogeneous multi-agent system. As explained earlier, an agent is a distributed computing entity and possesses autonomous execution control. It communicates with other agents through asynchronous messages, facilitating parallel and distributed computing. Behaviorwise, an agent may be reactive or proactive and may be mobile. Heterogeneous multi-agent systems involve a network of dissimilar agents cooperating with each other to achieve the overall system objectives.

The use of agent technology provides uniform interfacing mechanisms for integrating diverse computational components and means to incorporate intelligence into the control system. The software architecture is defined at two levels: operational level and application level. The operational level architecture defines components at individual processor level, while the application level elaborates means to develop distributed control tasks. Proper interaction between two levels is crucial in achieving distributed control. The following sub-sections describe the reference architecture of these two sections in detail.

5.4.1 Operational Architecture

The operational architecture describes the essential software components to operate hardware elements at controller level and to provide hardware independent interface to application level software. The operational architecture is local to every processor within a multi-processor intelligent controller. As shown in Fig. 5.4, the system level software components include I/O device drivers, hardware independent virtual device

interface, network device drivers, hardware independent network management interface and local agents of distributed real time operating system.

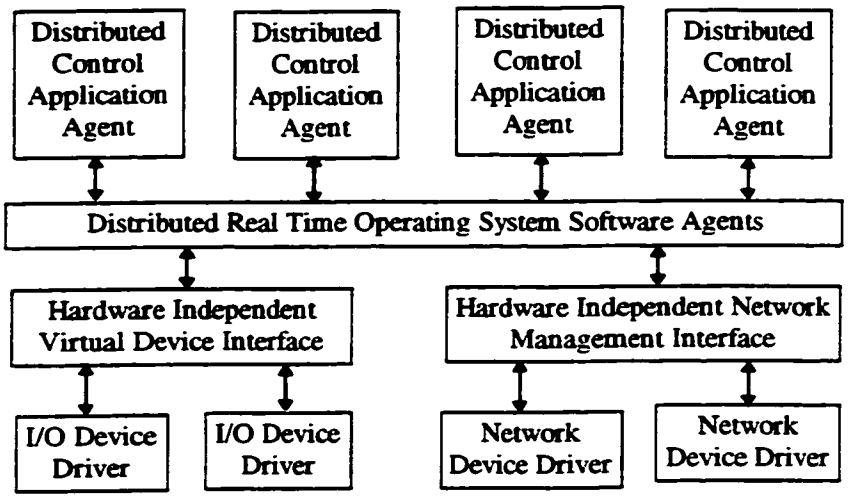


Figure 5.4: Operational Architecture

The I/O device drivers are hardware specific software modules that operate various hardware sub-systems within the controller such as process I/O modules and dedicated process/motion/logic control modules. The operating system elements interact with these drivers through a hardware independent virtual device interface. The network device drivers are network hardware specific software modules that operate local area network communication interface sub-systems and their processors. The hardware independent network management interface provides the link between operating system elements and network specific drivers. It also abstracts the mapping of physical and logical addressing mechanisms and routing of message packets.

The local agents of distributed real time operating system provide the core controller mechanisms to achieve distributed control. They provide resource management mechanisms at the local level according to global goals. Such mechanisms include processor scheduling and allocation, memory management, network scheduling and access, and I/O resource access. They also provide capabilities such as hardware independence, network transparent communications and enforce resource utilization based

on criticality. The application level distributed control agents use the facilities provided by system level software to achieve their goals.

5.4.2 Application Architecture

The application level distributed control architecture is comprised of reactive behavior based agents and these agents are specified using the IEC 1499 function block standard. This provides a single standard behavior specification mechanism for control agents of all manufacturing applications. The behavior based agents are specified using composite function blocks comprised of basic and composite function blocks, and the network of agents are distributed among intelligent controllers as shown in Fig. 5.5.

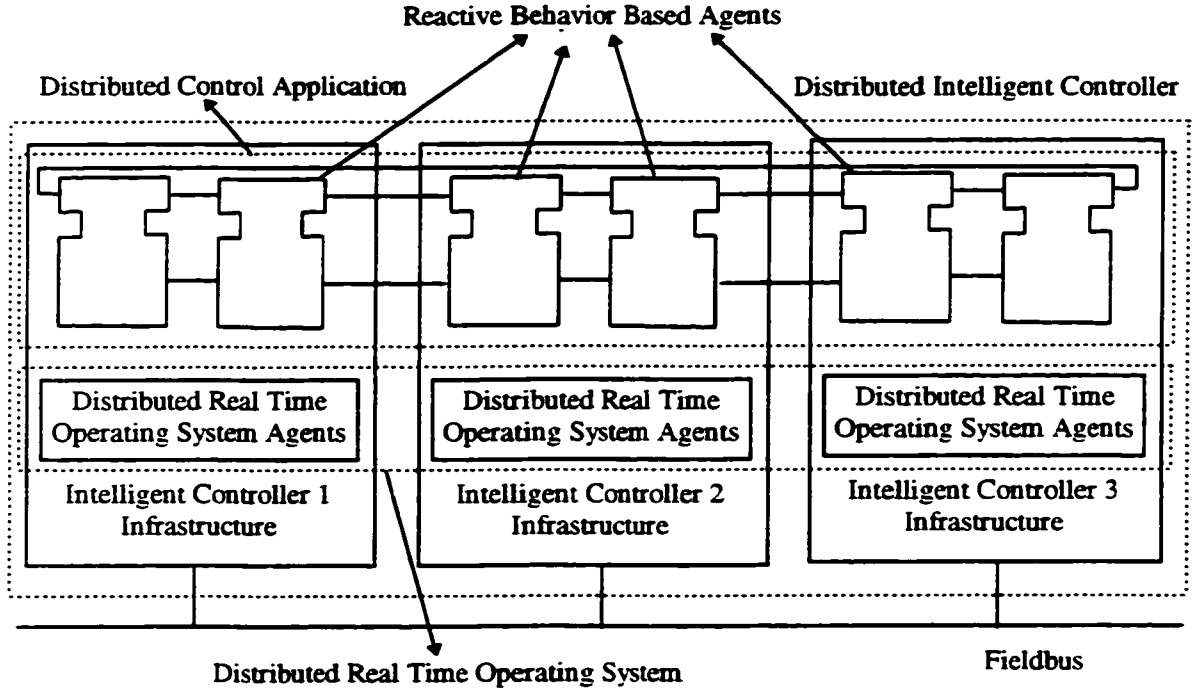


Figure 5.5: Application Architecture

The network of control agents, distributed on physically separate controller infrastructure, communicate and synchronize with each other through local agents of distributed real time operating system to achieve distributed control functions. The use of agent communication mechanisms lead to pass by value semantics even within same platform. Through the use of location transparency, the same agent network may be

configured in different ways for other applications. Further, the behavior based control results in control tasks being partitioned on behavior boundaries that execute in parallel.

The following section describes the functional architecture of an Agent based distributed intelligent controller.

5.5 Functional Architecture

The architecture described thus far lead to a functional architecture for the distributed intelligent controller as shown in Fig. 5.6 and has multiple layers. The distributed intelligent controller interacts with the machine/process environment and with other distributed intelligent controllers through the external physical I/O interface devices such as sensors, actuators and communication media. The distributed intelligent controller conceptually has two functional levels, the system level and the application level. The system level in turn has three layers, the hardware layer, the distributed operating system layer and the library layer. The application level also has three layers, the program layer, the execution layer and possibly the planning layer. Every layer in turn is comprised of multiple control agents.

The hardware layer is comprised of modular hardware entities that serve as the physical platform for agents. The distributed operating system layer agents provide a number of services to the agents of higher layers to interact with the lower hardware elements and the external world. They provide services to create, maintain, destroy and search agents of higher layers. They also help in separating hardware dependencies and provide location transparency. The library layer agents is comprised of reusable software in the form of standard Function Blocks and provide services to agents of higher layers or become part of them.

The program layer agents consists of user defined function blocks as local components of distributed control applications. The program layer agents also provide services for higher level agents facilitating information gathering and reconfiguration of operational strategies. The execution layer agents are involved in control tasks such as executing intelligent control strategies, fault diagnostics, fault recovery, data acquisition,

scheduling and decision making. The optional planning layer if present, would consist of agents that will be involved in strategic long term planning.

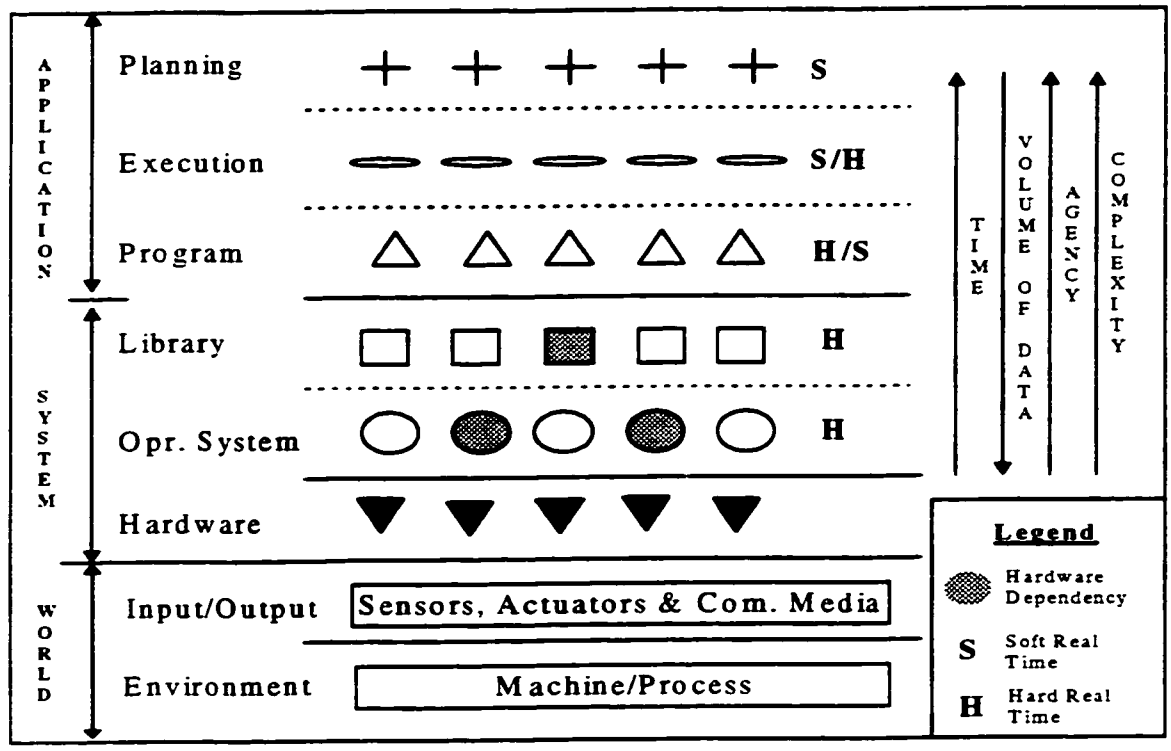


Figure 5.6: Functional Architecture

The lower layers, namely, operating system, library and program layers are predominantly hard real time since they have to respond immediately to internal/external events. On the other hand, the execution and planning layers are predominantly soft real time involving high variance execution periods. As one traverses down the multiple layers the following can be noted. The available time to respond, and the complexity of operations and hence the processing time decreases with lower layers. On the other hand the volume of raw data to be processed increases. Also the notion of agency i.e. the amount of intelligent responsibility delegated by the agents decreases progressively.

The means to engineer reference architecture described thus far involves a number of critical issues that need to be resolved. The following section discusses these issues.

5.6 Critical Issues

A distributed control environment is predominantly aperiodic and asynchronous event driven, having variable and unknown communication and computation latencies, being subject to overloads, having resource dependencies and conflicts. Despite such run-time uncertainties, the distributed intelligent controller must exhibit predictable timeliness for distributed trans-node applications. Hence it becomes the responsibility of system level software to provide adequate facilities and guarantees to meet distributed control requirements.

Distributed control intrinsically consists of multiple computations on multiple nodes which collectively perform an application that none could perform alone. This is accomplished by close, many-to-many, cooperation among peers through coordinated actions. However, since the computational agents are distributed, cooperation and coordination is accomplished through real time communications. This is equivalent to multitasking in a centralized multiprocessor, but with longer cooperation time constants due to inter-node communication latencies.

An ideal distributed control system is one that creates a single virtual centralized controller by providing single logical view of entire system. Inorder to create a single logical view, a distributed control system requires abstractions in terms of location transparency, network transparency, integrated priority based scheduled communications, flexible trans-node inter-task synchronization and communication mechanisms, distributed clock synchronization, fault-tolerance, and maintenance of high degrees of trans-node consistency.

Location transparency refers to the ability of an application software module to be independent of the target platform it will eventually be configured. It also refers to the ability to be independent of location concerns for its distributed counter parts. For example, as shown in Fig. 5.7, an application module may be configured for node 1 or 3 with its counter part at node 2. Because of location transparency incremental software development and reuse is facilitated. In the above example, application modules may be

configured twice on node 1 and once on node 3, one each for different application. Location transparency in conjunction with appropriate programming techniques also facilitates online dynamic reconfigurability.

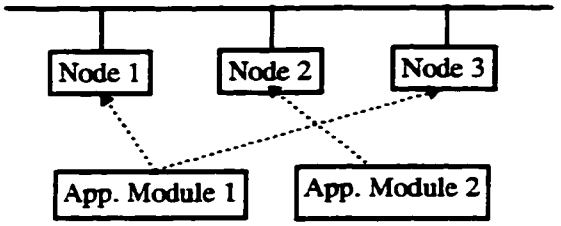


Figure 5.7: Location Transparency

Network transparency is the ability of application programs to be independent of physical network protocol characteristics. In other words, the system level software must abstract certain properties such as packet size, addressing, priorities, deadlines and medium access control from application level software. The system level software must also provide for multi-homed nodes with multiple types of networks. This would require sophisticated logical addressing and packet routing mechanisms. However, the application software is oblivious to the presence of such diverse components and interactions among them.

Maintenance of trans-node timeliness constraints requires prioritized and scheduled communication. It is important to distinguish between priority based message passing and prioritized scheduled communications. The former is found in most extant operating systems, and involves queuing/processing of messages according to their priority effective at local processor level. The latter prioritizes and schedules all real time network communications and the transmissions are scheduled on a global basis thereby satisfying distributed real time constraints. Scheduled communications ensure end-to-end timeliness constraints of trans-node applications will be met. Unscheduled communication such as a First In First Out (FIFO) approach will result in priority inversion, thereby violating distributed real time constraints.

Scheduled communications must also be integrated with system level software for proper reception. For example, consider the situation in which a particular task with a

certain priority is executing on the processor module of a controller node and a message is received by the communication module. If the message is meant for a higher priority waiting task, than the executing task, the communication subsystem must cause an interrupt signal in the processor module to preempt the executing task and let the higher priority task process the incoming message. On the other hand, if the message is meant for a lower priority task, then the communication module should not interrupt the executing task and the message has to wait in a prioritized queue till a suitable opportunity arrives.

Distributed control applications require efficient and flexible trans-node inter-task synchronization and communication mechanisms. Such mechanisms include binary event flags, asynchronous software interrupts, mutexes, semaphores, message queues. To accommodate communication requirements of real time and non real time tasks, various techniques such as time critical asynchronous publisher-subscriber, synchronous client-server and asynchronous client-server models of communications must be integrated into network management and system level software.

Distributed control requires a global time base to measure time instants at which events occur, intervals between events and to establish the causal order of events. Hence, the system level software must support synchronization of local clock with global clock and among other local clocks. Further, distributed control involves multiple failure domains. Examples of such domains include network overload or failure, failure of an I/O sub-system and software induced errors. The system level software must mask such failures from application programs and reconfigure alternative mechanisms dynamically.

Distributed or trans-node consistency refers to the maintenance of properties among nodes that is essential for correct system behavior. Examples of such properties include status information about trans-node resources and system level resources. Distributed control involves explicit maintenance of consistency using run-time facilities such as semaphores and locks. However, these facilities will have to be implemented with real time communication based message passing and hence the implementation must account for failure domains, medium access and communication latencies.

Because of such delays, deterministic inter-node consensus cannot be achieved instantly at any point in time and will take finite amount of time that can be bounded. Even though causal order of events may be established after a finite delay, the system mostly has to operate in an incomplete environment due to constant influx of new events. Hence, in a distributed control system, the usual problems associated with real time computing and distributed computing, such as scheduling, priority inversion and deadlocks get aggravated for the worse. Further, some of the traditional techniques that may be used in any of these systems alone are no longer applicable.

Conventionally, real-time control systems are variations of time triggered systems in which the task arrival rate and execution period are known a priori. The resource management for these systems is performed off-line by application programmers statically mapping the time constraints into fixed priorities. Such a simple approach is inadequate for dynamic distributed control environment. Dynamic systems are characterized by the asynchronous event driven nature, where neither task arrival times nor execution periods can be predicted.

In order to ensure timely response in such an environment, scheduling and resource management will have to be done in a dynamic manner. Further, autonomous control systems have differing requirements for computation and scheduling at various levels that are often conflicting with each other. For instance, the planning level has non/soft real time requirements, the execution level has soft/hard real time requirements and the control level has hard real time requirements. Obviously, the scheduling mechanism should handle computations with and without real-time attributes simultaneously to accommodate hard, soft and non real time computations.

The timeliness of a dynamic individual activity may be specified by a deadline time constraint for completing its execution. The timeliness of an individual activity is a scoped attribute of the activity i.e. timeliness is defined only within a scope. While executing an activity outside time constraint scope, its scheduling eligibility is based on attributes other than timeliness such as relative priority. Upon reaching the beginning of scope either through synchronous execution or asynchronous event, the scheduling must be based on

deadline constraint as well. The status remains unchanged until the end of scope or the expiry of deadline causing asynchronous exception.

Further, the dynamic environment introduces uncertainty about transient overloads and there exists a possibility of scheduling not being able to meet deadline constraint. Hence, a dynamic deadline failure handling mechanism is an important requirement for a distributed intelligent controller to support timeliness and complex interactions. Such a mechanism would facilitate remedial action to be taken in case of scheduling failure. The mechanism must be flexible enough for both synchronous and asynchronous execution scoping. Within a scope, the execution might be blocked due to reasons such as preemption or resource contention. The mechanism should keep track of completed execution and deadline periods to reschedule execution and to detect exception.

Metamorphic control system requires dynamic reconfigurability of all system components. This includes online changes at hardware level, network communications, system software level and application software level. Online changes involve ability to add, delete or modify components without requiring to shutdown the system and to recognize such changes immediately. For instance, online addition of a network communication hardware should be immediately accessible through installation of network driver software and transparent to the application level modules.

Given these requirements, the following sub-section identifies the limitations of extant systems to implement metamorphic control.

5.7 Limitations of Extant Systems

Existing real time operating systems have severe shortcomings for realizing metamorphic control. They do not offer many required services for distributed control. In addition, such existing systems would cause a major control bottle neck on this environment. With the exception of a few existing commercial real time operating systems, none of them were designed for distributed system application. Even though these exceptions offer location transparency they have severe limitations in other areas. Almost all commercial systems offer network connectivity and provide UNIX like pipes and

sockets.

These existing implementations use a TCP/IP protocol layer with FIFO queues for transmission control. They do not consider the relative priority of tasks nor message transmission deadlines for mapping onto communication priorities. There are no explicit communication scheduling mechanisms nor preemptive transmissions. In much the same way, communication reception is based on FIFO queues and system level software does not consider integrated message priority for reception. These features lead to unbounded priority inversion among distributed control tasks.

All existing systems assume loosely coupled interactions between application tasks and hence only provide synchronous client-server model of communication mechanisms. No system level support is provided for time critical publisher-subscriber communication model. Inter-task communication mechanisms are restricted to local processor level and similarly synchronization mechanisms do not span node boundaries. Global clock synchronization and fault masking are not considered at the system level. None of these systems make any effort to maintain explicit trans-node consistency.

Invariably, all commercial real time operating systems offer fixed priority based static scheduling. Static scheduling for distributed multiprocessor systems is not only sub-optimal but also violates real time constraints. Changing the priority dynamically during run-time does not solve this problem and at the best leads to unpredictable execution. Further, none of them consider deadlines and do not provide dynamic scheduling mechanisms. Hence the capability for detecting scheduling failures and executing remedial action does not exist.

In short, existing commercial systems are meant for unit level time triggered systems. They are not suitable for event driven distributed systems since they violate real time constraints and become potential bottlenecks in such situations.

A proof of concept prototype metamorphic control system suitable for the event driven distributed environment (such as holonic control) has been developed as part of this dissertation research is as outlined in the following section.

5.8 Prototype Metamorphic Control System

A proof of concept prototype metamorphic control system has been implemented by the author, through the development of a distributed real time operating system, a programming model and an implementation library, and a system engineering interface. Fig. 5.8 shows schematically, the components involved in this system. The physical infrastructure of this system used commercially available hardware for target platform and communications. The further details of the actual implementation are presented elsewhere in this dissertation (see Chapter 9).

The run time environment for the distributed control agents of this system comprises embedded target platforms with local components of the Distributed Controller Operating System (DCOS). The operating system provides location transparent system services to application programs and transparent services to interface with the communication network hardware. The application programs are created using graphical system engineering interfaces on one or more microcomputer hosts. The system engineering interface provides facilities to cross compile applications for the target platform, configure and maintain them, and to acquire run-time data.

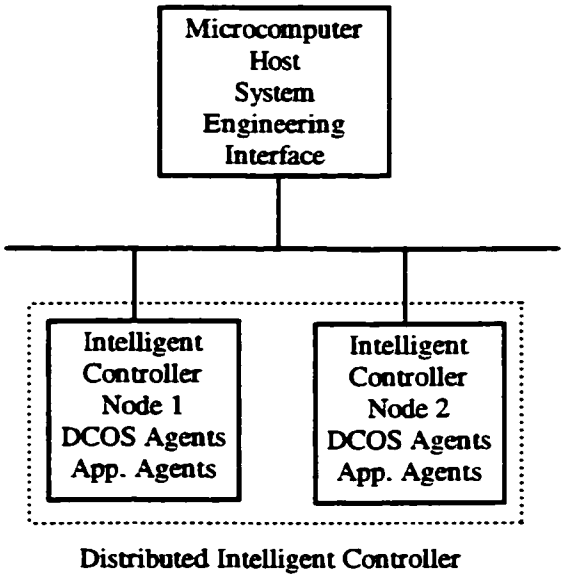


Figure 5.8: Prototype Metamorphic Control System

5.9 Summary

In this chapter, a reference metamorphic control architecture for distributed control of dynamic reconfigurable systems was presented. The architectural components at manufacturing system, physical hardware, operational software and application software levels were identified. The architecture is open and software centric, and uses a behavior based heterogeneous multi-agent system to provide a uniform functional model. The critical issues involved in the engineering realization of the reference architecture were elaborated and the limitations of extant systems were discussed. Finally, the components of a proof of concept prototype metamorphic control system which has been developed during this dissertation research were outlined.

Chapter 6

The Distributed Controller Operating System Design

6.1 Introduction

Operating System software should be efficient and flexible and desirably should survive a lifetime of changes without sacrificing either of these properties. Such changes include portability to new advanced hardware and on-line extension and/or modification of functionality for reconfigurability. The design of an operating system is central to achieve efficiency and maintainability. Object oriented design can efficiently address many of the issues associated with operating system development. An object oriented operating system can support the sharing of common interfaces and code, incremental extensibility, and the development of reusable software, by allowing methods which can take many different types of objects as arguments.

An object oriented operating system is one in which the components are organized as a protected dynamic collection of objects, defined by classes that are structured by inheritance. All components of the operating system, from low level entities like page tables and devices, to high level abstractions like memory regions and tasks are designed and implemented as objects. Interaction between these components is through the sending of messages between objects. The object oriented attributes of components are maintained dynamically across and within the privileged and non-privileged operating modes of the system.

In this chapter, the key concepts of operating systems and their design are introduced. This is followed by a detailed description of the design of the Distributed Controller Operating System (DCOS).

6.2 Operating System Concepts

In general, the purpose of an operating system is to provide programmers with an abstraction that simplifies the programming and management of a controller's resources. These resources include processors, memory, input/output devices and network interface devices. An operating system should control and manage resources reliably and efficiently, and often must enforce policies on their use. Fig. 6.1 shows the conceptual components of an operating system. The abstraction provided by an operating system is usually in the form of a set of primitive operations providing resource access and control. Programmers use these primitive operations, or primitives, when writing programs that need to obtain operating system services. The set of these primitives will be termed the operating system's application interface and programs using the services of the operating system will be termed application programs or simply applications.

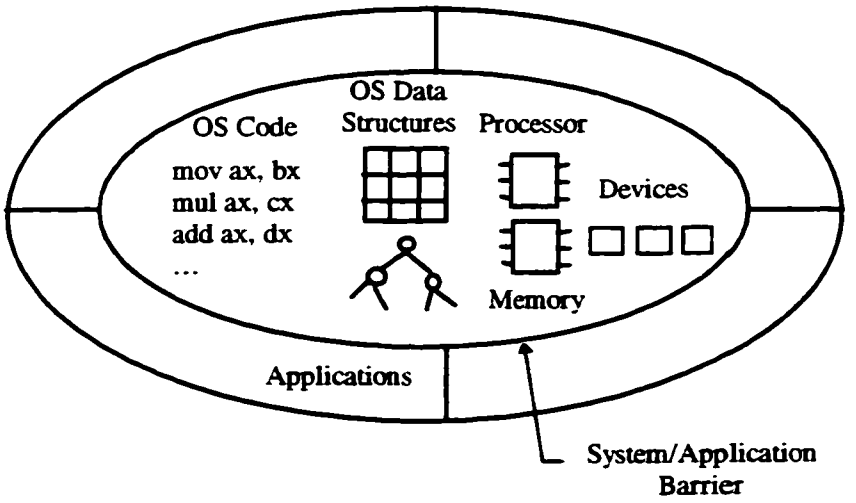


Figure 6.1: Conceptual Operating System

The complexity and richness of application interfaces provided by operating systems varies widely, but the common subset normally includes primitives to support device input/output (I/O), process creation/deletion, and memory allocation/deallocation. Examples of the low level resource management functions that an operating system must perform include handling interrupts from devices, or writing packets of data to a network interface. Even where these functions have direct hardware device support, it is the

operating system's responsibility to supervise such resource management functions from their initiation to their completion. Operating systems enforce policy decisions including scheduling priorities of application programs, the amount of memory allocated to each application, and how long each application shall be given the processor at a time.

Most operating systems impose a barrier between applications and system functions and data in an attempt to maintain integrity of their data and function, and ensure continuity of operating system services in the presence of potentially malicious or erroneous applications. This barrier is called the system/application barrier. The system/application barrier encapsulates the internal components of the system by limiting requests for system services to the operating system primitives. The primitives provide the only way to cross the barrier. Depending on the particular controller architecture and the requirements of operating system, the barrier may be enforced by hardware mechanisms, or be merely a programming convention. If it is simply a programming convention, malicious applications can circumvent the barrier.

The application interface primitives of an operating system allow delayed binding of application requests to the operating system functions implementing desired system services. Without such a delayed binding, applications would have to be linked together with the operating system, or include in their data addresses of the system primitive routines. The delayed binding permits the operating system to be changed without modifying the applications. Most operating systems provide their application interface primitives by an indirection through entry points into the operating system. The arguments to such entry points include the operation to be performed and the arguments to that operation. Such entry points decode the desired operation and its arguments, verify the arguments are correct, and call the proper operating system function that implements the service.

Avoiding corruption by application programs is just one part of the problem of protecting system data and keeping it consistent. Operating system software must also deal with potential inconsistencies in system data that the operating system software itself might cause. For example, if multiple threads of control are executing the operating

system code, they might leave shared data in an inconsistent state unless they synchronize their use of the data. Other potential inconsistencies can arise from interrupt processing. Interrupts can occur at almost any time during a system's execution. If an operating system is in the middle of updating system critical data and it receives an interrupt, the function to handle that interrupt might need access to the same data.

If the interrupted routine had not yet finished updating the data, the interrupt handling function might reference the data in an inconsistent state. Even worse, if the interrupt handling function were to further attempt to update the data, it could compound the problem by further altering already inconsistent data. The need to prevent potential inconsistencies in shared data caused by concurrent accesses is generally termed guaranteeing mutual exclusion. Protecting data from corruption during interrupts is usually addressed on a uniprocessor by disabling interrupts. On a multiprocessor, this is not enough; mutual exclusion is usually guaranteed by a combination of disabling interrupts to prevent accesses by the same processor and using spin locks and/or semaphores to prevent access by other processors.

6.2.1 Types of Operating Systems

The simplest type of controllers have no operating systems that is close to meaning of the term, and have no need of them. Programmers using such a controller have complete access to the entire machine. There is no protection mechanism to keep an application program from accessing any device or memory location desired. Applications for such controllers run sequentially. Each would reinitialize the machine for its use, consume its input, perform its computation and produce its output. A program for such a controller is, in a sense, simultaneously the operating system and an application. Operating systems for some embedded control systems behave this way . An embedded system is a control computer that is contained within a device whose primary purpose is usually not programmable control.

In the absence of an operating system, a programmer has to write functions to manipulate all the devices of the controller. These device manipulation functions are

difficult to write and must be duplicated in every application that uses the controller. For a simple embedded control system this is usually not a problem as there is often only one application ever written. But for controllers with multiple application programs reimplementing these functions should not be required for every application. If the functions are complex enough to warrant it, libraries of device control routines can be written and used by application programmers to avoid having to reimplement them for every application. These libraries could be considered crude operating systems or executives.

If a controller has many application programs to run, both processor and programmer time can be utilized more efficiently if each application does not have to be written to perform all the initialization and device management required. The solution to this problem is to create supervisor programs that initialize the controller and load and execute applications one at a time. When an application blocks/finishes, the supervisor program again takes control and reinitializes the controller for the next application. The supervisor also manages all the device I/O requests for the applications. A supervisor is implemented by localizing common I/O routines, leaving them always resident in the memory, and allowing application programs to call them. These routines define the application interfaces of such operating systems.

Many of today's unit level controller operating systems are designed this way. Most of these systems are, however, unprotected operating systems. Unprotected here implies that such systems lack hardware enforcement of their system/application barrier. Without hardware support for protecting the operating system functions and data from malicious or aberrant applications, such systems can be unreliable and easily corrupted. However, since existing controllers only offer restrictive and proprietary programming capabilities and tools, the lack of protection is not usually a problem. On the other hand, open architecture control systems allow unlimited capabilities with respect to programming which approaches that of a general purpose computer. In such systems, having to reboot and reinitialize the machine every time any application has a bug is unacceptable.

Modern processor architectures provide mechanisms that can be used by an operating system for protection. These mechanisms include privileged execution modes for processors and memory protection schemes. They prevent non-privileged application programs from accessing the memory storing the operating system data and functions. Operating system functions execute in the most privileged mode, allowing access to system data and functions. Application programs execute in the least privileged mode restricting them from accessing system data and functions. In addition, the least privileged mode prevents the execution of certain processor instructions that might compromise the systems security, for example, a set privilege mode instruction. The least privileged mode is also usually denied any direct access to resources.

Disallowing applications direct access to system memory and resources enforces the encapsulation of the hardware that the system provides. In such protected operating systems, primitives are implemented with special supervisor call or trap, instructions. These instructions raise the privilege level of the processor, and simultaneously jump to an entry point within the operating system's memory. Operating system entry points are functions that verify their arguments and then perform then requested service. Once the system service is complete, the operating system lowers the privilege level of the processor back to that of the application and resumes the application at the instruction following the call/trap instruction.

A multi-tasking operating system allows multiple applications to reside in the controller's memory simultaneously. The processor is shared by assigning it to an application with the highest priority or criticality at any given point of time. This gives more efficient utilization of a processor and its attached devices by overlapping I/O and computation. Applications must be protected from one another in a multi-tasking system. Multi-tasking operating systems must share controller resources (mainly memory) between multiple applications. Support for this is usually provided by allowing multiple address spaces that only the operating system can change between. Each application is assigned its own address space and cannot reference any data or functions in other address spaces.

Multiple address spaces can be provided in many ways, including: base/bounds registers, segmentation, and virtual memory.

As an alternative to building costly high performance processors, the proliferation of low cost microprocessors has allowed system designers to build high performance control systems out of a large number of small, inexpensive systems. Distributed real time operating systems support such a control system. Each controller, or node in a distributed system is connected to the others by some form of real time network allowing inter-node communication. A distributed operating system provides an application the abstraction of a single controller with all resources accessible through a uniform, location transparent mechanism. Many distributed systems exchange information between programming entities using messages. The entities are distributed across nodes in the system. Each entity has a global identification. Messages are sent to entities in such a way that they are independent of the entity's location. Operating system services are provided by such entities and may reside on arbitrary nodes.

In summary, operating systems range from simple embedded systems, to I/O library packages, to multi-tasking systems that protect themselves from applications and applications from each other, to distributed systems.

6.3 Operating System Design Techniques

Many design approaches have been applied to structuring operating systems to address associated software engineering issues such as extensibility, maintainability and portability. Most attack operating system problems by decomposing the system into smaller pieces, or modules, with well defined interfaces. Webster's dictionary defines a module as: any of a set of units designed to be arranged or joined in a variety of ways. Modularization is a major accepted technique to decompose and structure large software systems. The key to a successful modularization technique is to determine the correct granularity of these modules and to provide efficient data exchange between modules.

The following sub-sections give a brief overview of approaches to structuring and modularization of operating systems. These techniques have evolved in response to

advances in hardware technology and improvements in software engineering techniques. Each sub-section concentrates on a particular design philosophy and identifies its problems in order to motivate the design approach put forth in this dissertation.

6.3.1 Uninterruptable Monitor Approach

One of the earliest operating system structuring techniques is that of a single uninterruptable monitor program with a single thread of control. This type of system, in effect, dispatches or calls application programs in much the way that it calls functions within itself. An application program returns to the system under one of three conditions: an interrupt from a device requiring service, a service request from the program to the operating system, or the program's termination. Once the call to an application program returns, the operating system services the interrupt or request (or deletes the terminating program) and then resumes another (or possibly the same) program.

An operating system constructed as a monitor guarantees mutually exclusive accesses to system information since there is only one thread of control allowed to execute the operating system code at a time. While this thread of control is executing in the operating system, interrupts are disabled, prohibiting all but explicit changes of control. This makes implementation easier since the implementor can ignore such problems as mutual exclusion and concurrent access to system data structures. The main problem of such a design is the lack of scalability.

The frequency of calls to the operating system from interrupts and application service requests is proportional to the number of application programs and devices in the system. As the number of calls to the system increases, the time during which interrupts are disabled increases because the rising amount of time spent executing in the system routines. This in turn increases the number of interrupts that can be lost, or held pending for a long time, thereby decreasing I/O device throughput. A system structured to allow only one thread of control accessing system data must, on multiprocessor architectures, serialize simultaneous attempts to enter the system as the result of interrupts. This results

in a decrease in potential concurrent execution. Decreases in potential concurrency are also seen as application programs must wait for each other to enter and exit the system.

The monitor technique protects system data from interference, but at the price of scalability and performance. From a software engineering point of view, this approach has many problems as well. Such a system imposes no guidelines on how to structure its internals. Since all of the functionality of the operating system is placed within a single module, maintenance is severely impacted. The system internals are not divided into modules or sub-units that can be separately developed and maintained. This reduces portability and extensibility. These problems could be solved, however, with a good approach to further decomposing the monitor.

6.3.2 Kernel Approach

The kernel model of structuring an operating system is an attempt to remedy some of the performance problems of the uninterruptable monitor approach, in particular, those of scalability and device under utilization. It also attempts to further decompose the components of an operating system. This model treats all computational entities as processes, or threads of execution. Examples are application program processes, interrupt handler processes, and device driver processes. The kernel is primarily an inter process communication module. In the kernel model, an operating system is a set of concurrently executing system processes that request services from this kernel. Applications are likewise viewed as sets of concurrently executing processes that request services from the operating system processes via the kernel.

The kernel provides a minimal set of routines that perform the basic functions of inter process communication, process management, and interrupt processing. Higher level operating system functions are built around the kernel by using processes. The kernel is responsible for scheduling processes and directing interrupts to the proper system processes. Interrupts are disabled while executing within the kernel, but since most of the operating system functions are moved out of the kernel and into system processes, interrupts are enabled more often, thus improving device performance. A kernel should be

capable of processing multiple requests concurrently as long as the processes are programmed to use mutual exclusion to access system data. Therefore this approach is applicable to multiprocessor architectures.

A minimal kernel needs only to manage inter process communication and direct interrupts to the proper processes. Larger kernels may also create and delete processes, provide memory management, implement the application interface primitives, and supply a wide variety of other services. Kernels become more difficult to implement and maintain as they get larger. The cooperating, concurrently executing process model is the most valuable contribution of the kernel model. The extra concurrency provided by this model improves on the monitor approach. The idea of decomposing an operating system into a set of communicating and cooperating processes increases modularity thus aiding portability and extensibility.

The problem of identifying which processes should handle which operating system functions and further decomposing those processes, as well as the kernel itself, still remains. However, the kernel model remains the basis for the construction of most modern operating systems.

6.3.3 Layered Approach

Layered systems attempt to decompose an operating system by structuring it in small, easily understood, layers or levels. The processes or functions of the system are separated into layers that provide successive abstractions of the operating system. These layers are ordered by increasing level of functionality, and each layer depends only on the previous layer in this ordering. Usually, the hardware is the lowest layer, and the application interface is the highest layer. Many early layered systems divided an operating system into layers of processes performing system functions. This made it difficult to separate the logical activities of processes from the processes themselves.

In later designs, layers are built to reflect the functions in the system, thereby imposing a functional hierarchy. Various processes in a system invoke these functions, but processes are independent of individual layers. The lowest layer corresponds to the

hardware instruction set of the processor. Functions in higher layers can use functions from lower layers. Concurrent processes within the operating system can access functions at different layers within the hierarchy. Layering aids in implementation, debugging and testing of the system. Layers enhance portability; if lower layers hide the hardware, only these layers need to be changed when retargeting the operating system to new architectures.

An implementor can ignore the implementation details of lower layers but still use their functionality when designing and debugging higher layers. This improves maintenance. For example, some layered systems use as their lowest layer the concept of an abstract machine representing an idealized computer architecture. This reduces the portability problem to reimplementing the abstract machine for the available computer hardware. This also allows multiple virtual computers to be simulated on a single physical computer by supporting multiple concurrent copies of the lowest layer each sharing the physical computer. Operating system software can be developed and debugged on any of the virtual computers and, when ready, be run directly on the physical computer without any changes.

A major difficulty with building layered operating system kernels is determining the layer in which a process or function belongs. Since each layer may only rely on the processes or functions provided by lower layers, careful planning is necessary. Another problem with layered systems can be performance. If a layered system is structured in such a way that a layer has access to only the layer directly beneath it, performance can suffer as requests must traverse several layers to achieve a low level service. It is more desirable to allow a layer to access the functionality in any of the layers beneath it.

Perhaps the biggest drawback to layered systems is that the granularity of the abstractions it provides (the layers) are too coarse. However, layering is orthogonal to many other structuring techniques. When further decomposed into servers, as in the message-passing approach or into the objects in object-based system as discussed in the following sub-sections, layers can substantially aid the documentation and high level understanding of a system.

6.3.4 Message Passing Approach

Message passing operating systems are systems based on the kernel idea. They attempt to further decompose an operating system's structure. Message passing systems use explicit send and receive operations to exchange information (messages) between concurrently executing processes. Each of these processes (or often sets of processes) is usually viewed as a server providing functionality to other client processes. References to servers are obtained from name servers, which convert symbolic service names to references to servers implementing the service. Each message from a client includes a request to the server and arguments specific to that request.

In a message-passing system, all communication and computation is achieved by explicit message exchanges between clients and servers. Messages are sent to servers and replies are sent back. This message exchange may be synchronous, in which case the sender does not continue executing until the reply is received, or asynchronous, in which case the sender continues to execute and awaits the reply whenever desired. Processes executing on behalf of application programs are often just consumers of services and may not provide any of their own.

In message-passing systems the kernel is usually viewed as a server as well. It can be composed of many processes; each of which executes on behalf of the system to perform system management functions, for example, handling interrupts and creating or deleting new processes. Processes desiring a service from the kernel send a message and (optionally) await a reply just as they would do if requesting a service from another process. Message-passing systems come in two forms, those that consider all message receivers to be processes or servers directly, and those that consider message receivers to be message ports read by servers.

In a message-passing system using ports, server processes poll selected ports when ready to receive a message. In the other type of system, messages are sent directly to a target process and are received the next time that process executes an anonymous receive primitive. The receive is anonymous in the sense that the process simply receives the next

message queued for it. In a port-based system, the server process could selectively chose which port to receive the next message from, giving more flexibility in assigning priorities to messages. Message-passing systems work well in distributed environments. The only support needed is to provide server identities that can be used independently of location and message send and receives that can span machine boundaries. In this way, processes on one machine request services on another machine in exactly the same way they request services on the machine they are executing on: by sending a message and awaiting a reply.

Decomposing the operating system into a set of servers increases both portability and maintenance. Portability is improved since only servers relying on machine specific details need to be retargeted for new architectures. Maintenance is assisted by the decomposition of system functions imposed by servers. One problem with such systems is that a message send/receive is usually much more expensive than a normal procedure call. Since any message send can potentially cross a machine boundary, arguments must often be copied rather than being referenced of the stack of the sender of the message. Likewise, the synchronization between the sender and the receiver imposes additional overheads. For example, a context switch may be incurred from the sending to the receiving process.

Many system designers have gone to considerable extents to minimize message sending costs. Having one process execute on the behalf of multiple servers can reduce context switching costs. Since a high level abstraction likely makes multiple requests on a low level abstraction, requests to low level abstractions are usually more common than those to high level abstractions. Therefore, rather than starting with a complex scheme for handling high level abstractions that will not easily and efficiently handle the low level cases, it would be desirable to start with a simple scheme for handling low level abstractions that is efficient and will scale up to higher level abstractions. This allows a single paradigm to be used throughout the construction of the system.

6.3.5 Object Based Approach

Object based approaches to operating system design replace the kernel model of communicating, concurrently executing processes, with a collection of communicating,

cooperating objects . Each object in the collection represents a particular logical entity of the system. Objects can represent processes, memory ranges, communication channels, devices, and many other operating system abstractions. Each object provides a set of operations available to other objects in the system. These operations define the behavior of the object and the interface provided to other objects.

In this model, objects invoke such operations by sending messages to other objects. These messages are conceptually similar to the messages in the message passing approach. The main difference is that sending a message to an object is always synchronous and no explicit receive is needed. This encapsulation of behavior in object-based systems closely parallels the software engineering concepts of modular programming and data encapsulation. An object can send a message to any other object as long as it has a reference to that object. Like the servers in message passing systems, objects can reside on different nodes in a distributed system.

Object based approaches are more data-driven than message passing approaches to operating system construction. They separate the abstractions of a system into different modules (objects), each with a well defined function and interface. Objects address other objects in a system by means of a reference or capability to the object. The reference can define what permissions the invoking object has with respect to the object on which it is operating. These references can be as simple as direct pointers to other objects, in effect providing no permission enforcement, or as complex as capabilities in full protected capability based systems. In a protected capability based system, only trusted objects can modify and distribute capabilities. Some systems provide hardware support for this protection, while others rely on indirection through a trusted manager of capabilities.

Object based approaches address many of the operating system engineering problems efficiently. Sets of objects can be used to abstract the hardware and thus increase portability. Objects also represent a small enough encapsulation to improve maintenance and documentation. The main potential problem of object-based systems is, like message passing systems, one of efficiency. Efficiency in an object based system is a function of the expense, or weight, of objects and the implementation of message sends between objects.

Object based systems span a spectrum of implementations. At one end of this spectrum are systems that use the object/message send paradigm to structure the servers of message-passing systems. Objects encapsulate servers and object messages structure the messages exchanged between clients and servers by automatically providing the operation code and defining the types of parameters. The object interface gives structure to the interface that the server presents to its clients and defines the messages sent between the client and server. However, such systems suffer the same performance penalties as message-passing systems.

At the other end of the spectrum are systems that use an explicit send/receive paradigm to implement message sending, transfer control between objects by explicitly weaving threads of control from object to object. In such systems, objects are usually passive. Message sending is implemented with traditional procedure calls. The thread of control of the invoking process enters the object to perform the operation. This reduces synchronization and context switching costs and involves no added expense of argument copying. The one problem with such systems is that, without the underlying message send/receive paradigm, they require extensions to handle distributed cases.

Object based systems address the software engineering problems for operating systems by decomposing its functionality into small modules (objects) with well defined interfaces. Since objects and message sending can be made efficient enough, performance is not a problem. Hence the DCOS architecture uses primarily an object based approach to operating system design and is discussed in detail in the following section.

6.4 DCOS Architecture

The DCOS is a distributed real time operating system that provides deterministic execution for multi-sensor based dynamic reconfigurable control systems. In theory, it can provide location and network transparent services for applications distributed across several thousands of heterogeneous processors. The system can be a loosely coupled network of heterogeneous real time sub-networks under arbitrary hierarchy and the sub-networks themselves being a loosely coupled system of heterogeneous uniprocessor and

multiprocessor sub-systems. It provides several features for autonomous metamorphic control such as dynamic scheduling, integrated priority based scheduled communications, dynamic reconfigurable applications and online extension/modification of functionality.

DCOS is a preemptive multi-tasking system for 32 bit processors and the application tasks may be "heavy weight" processes or "light weight" threads. The operating system provides explicit process and thread model of task management. A process specifies the virtual address space boundaries within which multiple threads of execution control can coexist. A thread represents an execution unit with separate stack and task management data structure and is the individual unit of schedulability. The threads within a single process can communicate directly with other through shared memory. A process with only one thread of execution control is the equivalent of a conventional heavy weight process.

The DCOS uses a layered object based architecture and Fig. 6.2 shows the local components of distributed operating system. The architecture is comprised of four layers with each layer further being sub-divided into system level agents, system support tasks and user applications. These agents assume an abstract hardware that can be accessed through interface objects. The actual implementation of interface objects/methods has the hardware dependent components. Layer 0 is comprised of system and scheduler agents. They provide core hardware independent functionality to manage controller hardware. The scheduler agent is associated with managing the processor, while the system agent is responsible for structured management of device hardware.

Layer 1 is comprised of timer agent, task agent, buffer pool memory agent, segmented heap memory agent, message port agent, distributed shared memory agent and semaphore agent. The timer agent provides hardware independent facilities to manage system timer and clock, and virtual software timers. The task agent provides capability to manage application level tasks with simple synchronization mechanisms. The buffer pool memory agent and segmented heap memory agent provide flexible high level memory management mechanisms for fixed and variable sized buffers respectively. The message port agent provides client server type inter-process communication and synchronization

capability. The distributed memory agent provides primary mechanism for publisher subscriber model of communications. The semaphore agent provides location transparent resource management and synchronization capability.

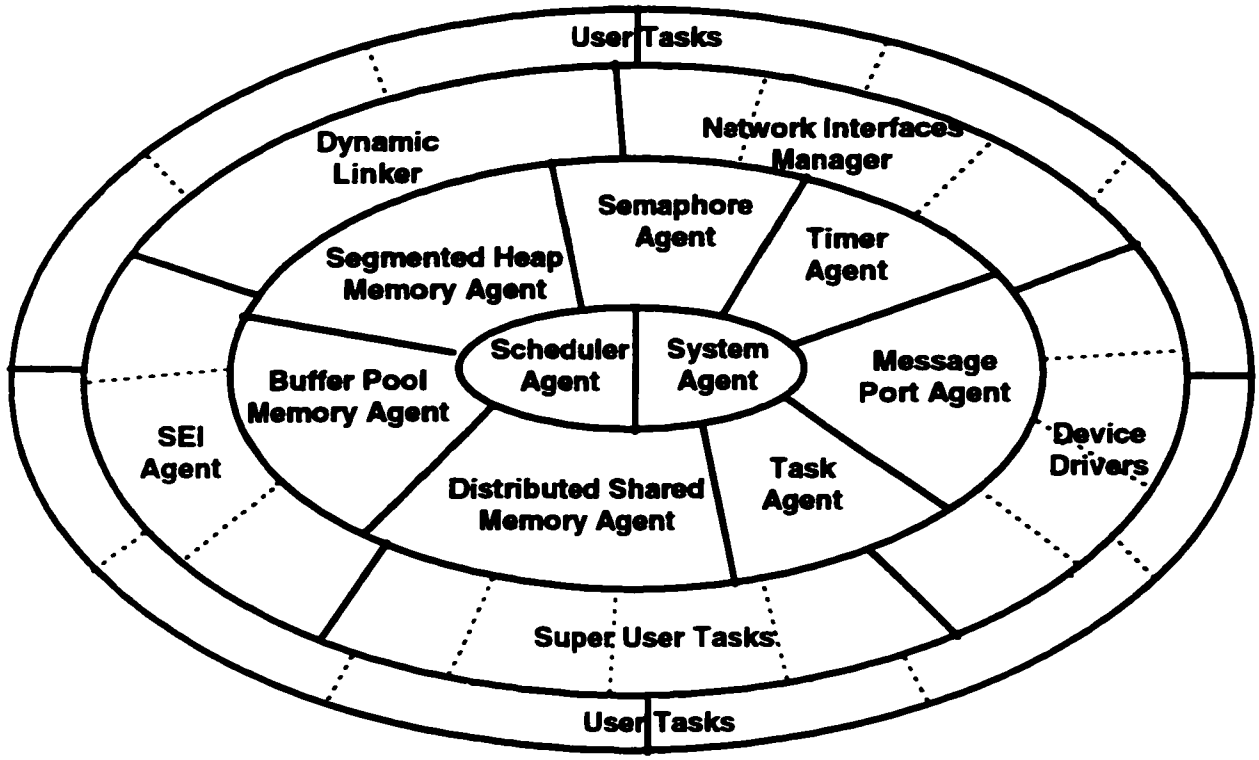


Figure 6.2: DCOS Architecture

Layer 2 is comprised of device drivers, network interfaces and manager, dynamic linker, system engineering interface agent and super user tasks. The device drivers are hardware specific software modules that operate special purpose hardware such as I/O and motion control. Unlike conventional operating systems, software device drivers are not statically linked into the kernel. Instead they can be installed and removed online thereby providing dynamic reconfigurability. The network interfaces provide access to physical layer of various network devices through appropriate network specific protocol stacks. As with the device drivers, network interface software can be added or removed dynamically online. These protocol stacks are integrated by a network interfaces manager that also provides capabilities such as mapping of logical to physical addresses and network routing.

The dynamic linker provides the capability to incrementally load, relocate and link new binary code modules and their symbols. It also provides facilities to remove such modules dynamically. Unlike conventional operating systems, every process does not need to have its own copy of executable code. The dynamic linker facilitates code sharing among various processes thereby reducing memory requirements. As the name implies, the system engineering interface agent acts as an agent on behalf of remote system engineering interfaces distributed across microcomputer hosts. It coordinates remote management commands from system engineering interface, and alarm and data logging requests from system and application tasks. Remote management commands include addition/deletion of new system and application software modules, creation/deletion of system and application tasks and configuration of operating system resources.

Super user tasks are administrator defined tasks that run with special privileges as compared to user tasks that run at Layer 3. As may be noted, though there may be multiple super user tasks, there is only one super user process. The system/application barrier is imposed between Layer 2 and Layer 3. Therefore the user tasks cannot access system data structures and resources directly and have to use system primitives to access them. Further, certain system services are considered privileged. Such services include ability to configure operating system resources, add/delete device drivers, network interfaces, logical to physical address information, and binary code modules. Such services may only be executed by Layer 2 tasks such as super user tasks and system engineering interface agent tasks. Since access to these tasks can be secured through an authentication procedure, the operating system provides capability for security critical and safety critical operation.

All the system agents are multi-threaded active objects that communicate with their distributed counterparts through message passing. In a conventional server based message passing operating system, every service request involves a send/receive operation. This operation in turn involves a context switch with priority inheritance by the server task. This feature results in costly system services. Further, preemptibility is affected since simultaneous service request by a higher priority task will have to be processed

sequentially. In order to avoid this undesirable feature, the object based system agents are multi-threaded. This removes the need to inherit priority from sender and sequential processing among senders. However, each thread of a system agent executes on a separate stack from the user stack to protect system integrity. Though this involves a partial context switch, it is not as costly as a full fledged one. This also means all system agents execute under multiple contexts which introduces the need for proper synchronization and atomicity of operations among threads to maintain integrity of system data.

In the case of a remote service request that needs to be accomplished through message passing, special mechanisms are used and as shown in Fig. 6.3, typically, the following steps are involved. Once a system agent identifies that a remote operation is required on a remote object it constructs and transmits a remote request message with required parameters on behalf of sending task. The sending task now blocks depending on the type of transaction requested. Upon receiving a request, the remote system agent performs the requested operation on appropriate object and constructs and transmits a response message with results of operation. The system agent that sent the request receives the response message and copies the result to the blocking task and unblocks it for execution. All of this procedure happens transparently to the user task thereby providing location and network transparency.

Whenever a remote operation is requested, the system agent also sets a watchdog timer, so that an unreachable or erroneous transmission can be detected. If such an error is detected, an appropriate error code is returned to the requesting task. This behavior of sending and receiving messages by system agents is somewhat similar to that of servers in a conventional message passing operating system. It should be noted that the system agent on remote node processes the requested operation on a separate stack from those belonging to local tasks in that node, to maintain system integrity. Because of this multiple simultaneous requests are processed sequentially according to their relative priority. The priority of a request is transmitted along with the request and is inherited by receiving agent to allow for preemption by local tasks. The priority of a request is the priority of requesting task.

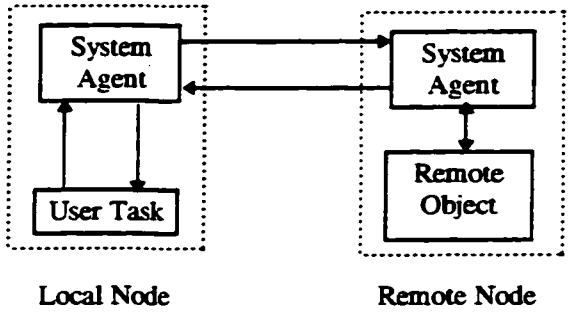


Figure 6.3: Remote Transaction

The operating system also provides integrated priority based communication scheduling. This mechanism significantly alters the way in which any communication is transmitted or received. For instance, whenever a remote operation is required the priority of the requesting task is mapped onto the network priority for communication. All pending transmissions for a particular network interface are processed according to their relative priority. Similarly, on the receiving end an incoming message is allowed to interrupt execution of a local task only if its priority is higher than the priority of executing task. Otherwise, the request message is queued for processing till suitable opportunity arrives.

Being an object based operating system, all the entities of system are represented by objects. Examples of these system objects include tasks, semaphores, message ports and memory regions. The system agents provide a number of services to create, modify, manipulate and delete specific system objects. In fact all system service primitives manipulate system objects. Being a distributed object based system, it is necessary to provide location transparent mechanism to manipulate objects. For this purpose and to maintain integrity direct access mechanisms such as pointers cannot be used. Hence an indirect logical object addressing mechanism is used to reference all system objects. The logical identifier of an object is required by all service primitives thereby providing location transparency.

All system objects are assigned a logical object identifier that is unique throughout the distributed system. As shown in Fig. 6.4, the logical object identifier is 32 bits in length and has three major components: a node identifier, an type identifier and an object identifier. The node identifier is 16 bits wide and represents the logical address of the node

in which the object resides. The type identifier is 4 bits wide and denotes the system object type. The object identifier is 12 bits wide and represents a specific object. Together, these three components provide a way to uniquely identify a particular object. The logical identifier of an object is returned by any primitive that creates an object. Further, the operating system provides a user assigned naming mechanism for system objects. Hence, it is also possible to obtain logical object identifier at any point during an objects life through its name. All the system agents provide dynamic name to logical object identifier look up service.

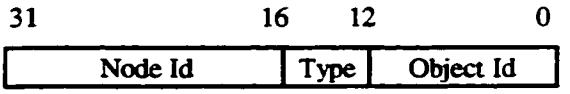


Figure 6.4: Logical Object Identifier

The 12 bit object identifier is used by system agents to locate actual objects efficiently through a table look up mechanism. The table is an array of pointers to locations of system objects in groups of eight. Hence, system objects can only be added in multiples of eight during DCOS configuration. The agents maintain tables by filling them with valid pointers during configuration and system objects with flags indicating whether they have been used. This feature helps in identifying an invalid object identifier. All allocated system objects also have a pointer to their owner task, so that the legality of an operation requested by a task can be verified.

It should be noted that barring few exceptions, a task can request an operation on a system object only if it owns it. Ownership under normal circumstances belongs to the task that created object, but can be transferred explicitly by one task to another through transfer ownership primitives provided by all system agents. All system objects owned by a task are considered as system resources allocated to this task and a list of all such resources is maintained. If a task fails to return certain system objects through delete primitives, before it terminates normally or abnormally, the system recovers all such resources automatically. This also includes address space and physical memory. Combined with protection, this resource recovery feature makes DCOS stable in the presence of

erroneous applications.

The system objects can be created with a visibility attribute that defines the visibility of an object in the system. The visibility attribute can be one of local, regional or global. A local object is visible only within the node it is residing. A regional object is visible only within the nodes of a distributed intelligent controller. A global object is visible on all nodes throughout the factory control system. The visibility of an object determines whether a name lookup service in a remote node will reference this particular object. Depending on the visibility attribute, the distributed database throughout the system is dynamically updated, whenever an object is created or deleted. It is the responsibility of user to define a unique name for every system object to accomplish unique name to logical identifier mapping.

The service primitives provided by system agents are classified into two categories: configuration services and real time services. The configuration services are used to configure operating system such as adding a number of system objects that could be instantiated by user tasks later. The real time services are used during run time by application tasks for control. All real time tasks execute in a deterministic manner with O(1) cost, while all configuration services execute with a varying cost of O(N). The notation O(1) is used to denote deterministic execution with a constant computational cost always, while $O(\log_2 N)$ or O(N) is used to denote that the worst case computational cost for the operation is proportional to $\log_2 N$ or N, respectively. Irrespective of service type, all primitives are preemptible while in kernel mode and hence DCOS provides deterministic performance.

6.5 Summary

In this chapter, the concepts of an operating system and its design were reviewed. After discussing various design strategies, an architecture that is primarily object based but incorporating some features from other design methods was presented. This architecture was used to implement the distributed controller operating system with metamorphic control capabilities and is discussed in the following chapter.

Chapter 7

The DCOS Implementation

7.1 Introduction

The DCOS is an object based distributed real time operating system that integrates important operating system techniques such as virtual addressing and paging, fully reentrant kernel, multitasking through abstractions of processes and threads, support for multiprocessors, integrated network management and location transparent distributed services. The system is distinguished from other distributed real time operating systems by its implementation as a protected, multi-programmed distributed real time operating system for high speed reconfigurable multi-sensor based control systems. It is based on object oriented techniques and provides an object oriented application interface to access system services.

In this chapter, the details of DCOS implementation are discussed. The various agents, their service primitives and algorithms used are also presented.

7.2 Implementation Details

The programming language used to implement an object-oriented operating system can drastically affect its performance. The implementation language chosen for DCOS was C++. This is mainly due to the advantages of statically typed object-oriented languages, along with the added advantage that, although they violate the pure object-oriented paradigm, being a superset of systems language C, C++ allows certain low-level programming techniques necessary for the easy and efficient implementation of an operating system. In particular, the language allows the programmer to specify an object's representation in memory, to place objects at a specific address, to predetermine the size of an object and to use inline assembler directives to directly manipulate hardware.

Specifying an object's representation in memory is necessary to allow classes to represent hardware defined entities such as device or processor control registers and

device command/control messages. It is also necessary in order to allow data structures specified by certain standards, such as the representation and the placement of fields in a network protocol packet, to be encapsulated within objects that are instances of representative classes. The ability to specify a new object's location in memory is necessary to allow addressing hardware specified entities as objects once representative classes have been designed. Again, this includes entities like device registers or hardware defined data structures that are often at a fixed location in memory. Finally, the ability to precisely determine the size of an object is useful to optimize memory allocation/deallocation for frequently instantiated classes.

C++ does not always faithfully implement the object-oriented paradigm, mostly due to the backward compatibility maintained with C. It can, however, be used as an object-oriented language and allows an examination of the advantages of object-oriented programming applied to operating systems. C++ supports objects, classes, single and multiple inheritance, and polymorphism. However, every value in a C++ program is not an object, in particular, primitive types such as integers, floating point numbers and characters are not objects that are instances of representative classes. This is a concession to simplify code generation and optimization. Having such primitive types built into the language allows a compiler to generate traditional C equivalent code for operations on such types. Specifically, since no method lookup is done for such operations straight one-to-one mappings to machine code exist and can be expanded inline in the generated code.

Another violation of the pure object-oriented programming is that C++ allows direct accesses to instance variables. Perhaps the biggest drawback of C++ is that it implements polymorphism only in the form of inheritance polymorphism and does not support other types of interface signatures. This is mostly a concession to efficiency and, along with static typing, allows C++ to implement a very fast method invocation scheme. However, it forces the programmer to constrain the type hierarchy to the class hierarchy i.e., a concrete class implementing a desired signature must be a subclass of the abstract class defining the signature.

7.2.1 Class Design

Fig. 7.1 shows the major classes used in the design of DCOS. As may be noted, these classes fall into three distinctive sets. The first set is composed of classes that implement basic and supporting functionality of DCOS. Classes Single Link, Singly Linked List, Double Link and Doubly Linked List implement basic list management functionality. Classes Segment, Ram Heap, Memory Manager and PMMU manager implement low level paged memory management capability. Classes Packet, SEI Packet, Message Buffer, Address Entry, Route Entry, Protocol Interface and Network Manager implement low level communication and message passing capability. Classes Process Context, Device, Async Info, Clock, Timing Info, Wait Info, Owner Info and Object Id implement miscellaneous functionality required for system object and task management.

The second set is composed of classes that implement system objects, while the third set implements functionality of system agents. OS Object is the super class of all system objects, while OS Agent is the super class for all system agents. The individual system agent is a friend class of its respective system object. For instance, System agent is a friend of HW Device object, Task agent a friend of Task object and so on. Together system agents and objects implement the requisite functionality of operating system in a modular fashion. Hence, some of the OS functionality can be readily omitted by removing appropriate classes. For instance, only system, scheduler, task and timer agents are required for a minimum configuration.

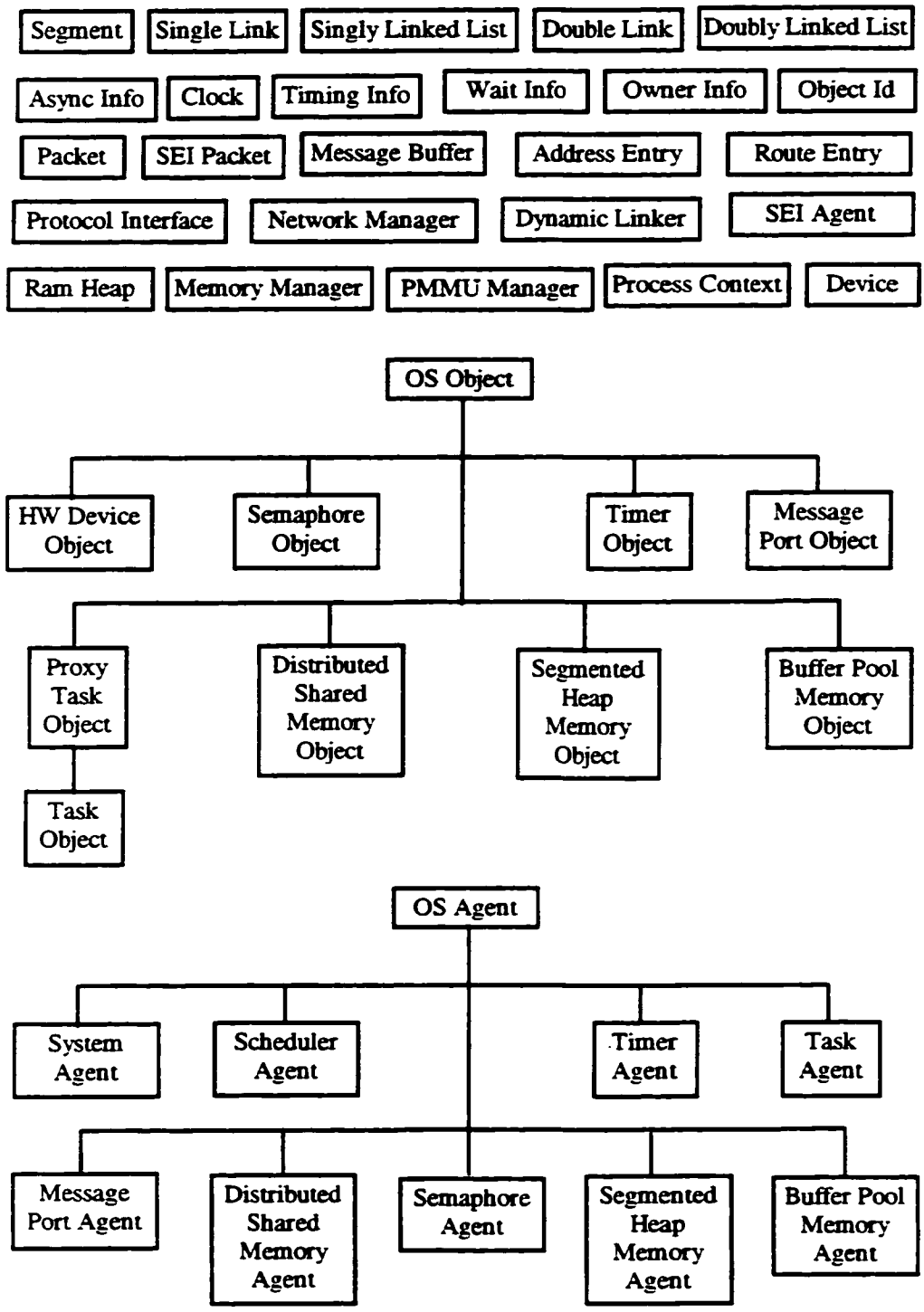


Figure 7.1: Class Design

7.2.2 Virtual Memory Management

DCOS implements memory protection mechanisms using the virtual memory management capabilities of modern processors. However, DCOS does not provide any demand paged virtual memory management. This is so since disk file systems are not typically present in controllers and swapping to file leads to unpredictable execution. Virtual addressing and paging mechanisms are used to implement process address space. Fig. 7.2 shows the memory partitions on a typical 32 bit processor providing 4GB of address space. The 4GB address space is halved to implement a 2 GB per process data segment that is read/write accessible to an user level application.

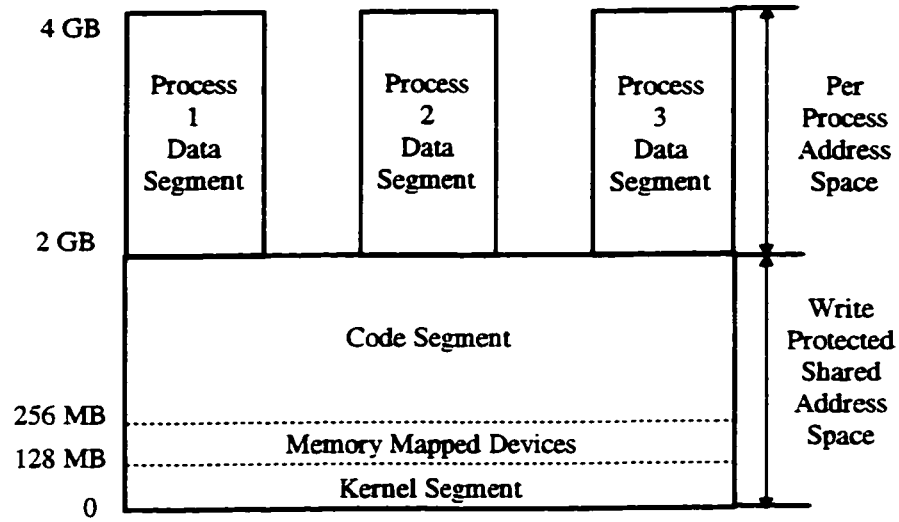


Figure 7.2: Virtual Address Space

The lower 2GB address space is shared by all the processes. This address range is write protected from applications and is further sub-divided into three regions. The lower 128MB is set aside for kernel segment where the operating system code and data reside. The next 128MB is set aside for memory mapped I/O devices and the rest is set aside for code segment. The code segment is the region where code for all processes reside. DCOS provides dynamic code loading and linking capability and both the binary code and online symbol table is shared. This results in significant savings towards memory requirements.

Fig. 7.3 shows a typical two level paging mechanism used by modern processors. The individual page directory, page table and page sizes vary among processors along

with the hashing and lookup mechanisms. However, the typical logical to physical address mapping involves following steps. An upper portion of the logical address is used to identify a page directory entry in the page directory for current process. This leads to a page table that is used in combination with middle portion of logical address to identify the physical memory page. Finally, the lower portion of logical address is used to offset into physical page to access requisite location. Most of this address translation process occurs in hardware though it is the responsibility of operating system to set up appropriate translation tables.

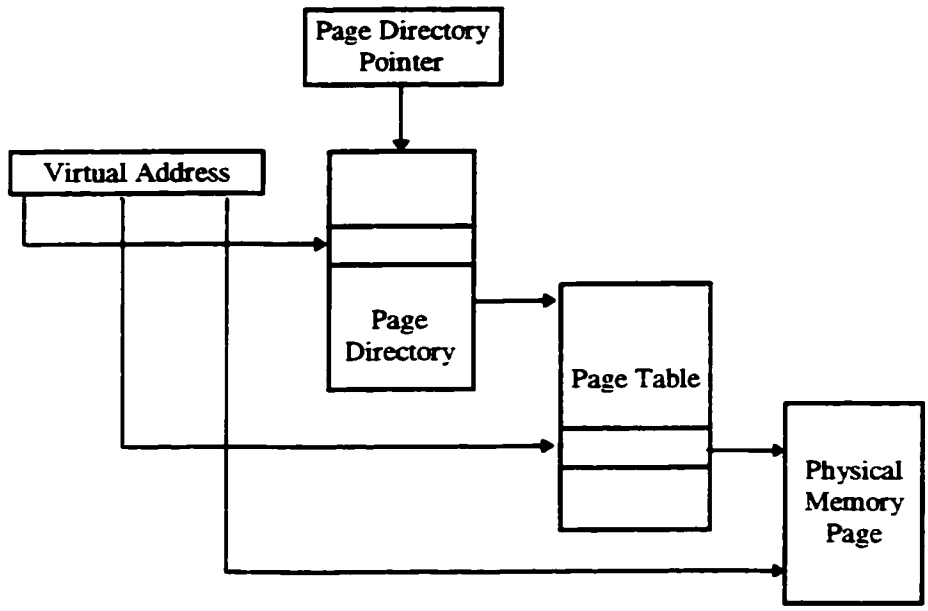


Figure 7.3: Virtual Address Translation

7.2.3 Hardware Dependencies

As mentioned earlier, the system agents assume an abstract hardware layer that is accessed through interface objects and methods. These interface objects/methods implement hardware dependent code to manipulate specific sub-systems such as priority interrupt controller and programmable interval timer. Other hardware specific code include context switching, floating point processing, bit manipulation, device I/O and multi-precision arithmetic and comparison operations. The inline code generation facility

of C++ is used to efficiently integrate such code into operating system without any overhead.

7.2.4 Application Interface

The application programming interface for system primitives of DCOS is also object oriented. However, since the DCOS is a protected mode operating system imposing a strict system/application barrier, the user tasks cannot access system agents or objects directly. To circumvent this situation dummy objects are used. As shown in Fig. 7.4, these dummy objects correspond to system agents on application side. The user tasks invoke system primitives as methods on these dummy objects in usual manner. These dummy objects in turn raise the execution privilege by jumping to a system trap and in the process switching stacks.

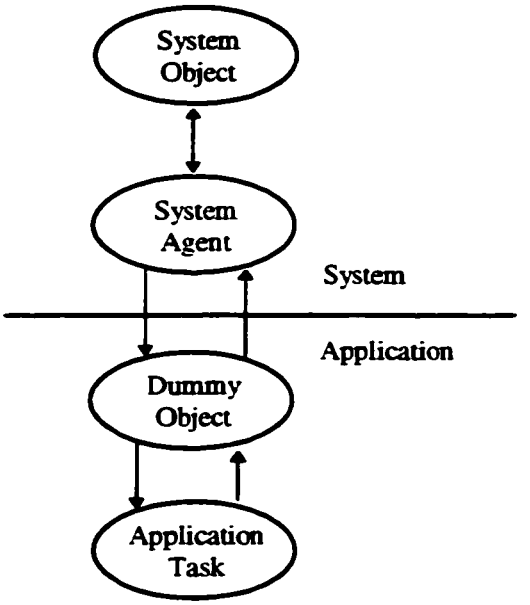


Figure 7.4: Application Interface

The system trap performs a high speed message dispatch to appropriate system agent that checks the arguments and performs requested service. The result is returned to the dummy object and the execution privilege is lowered along with stack switch in reverse. The dummy object in turn returns a result to the user task for continuation. The dummy objects are part of application software development libraries. This interfacing

technique uses normal C++ method invocation mechanism on objects and as a result makes use of static type checking capabilities of compiler. It is also an efficient and low overhead method for crossing protection boundary.

7.3 The System Agent

The System Agent provides a structured way to manage I/O device sub-systems. It also provides facilities to manage external interrupts to the processor. It provides a number of services to manipulate device hardware and software and their associated interrupts. Due to protected nature of DCOS, all I/O instructions and memory accesses are subject to privilege checking. Similarly, interrupt servicing software cannot be directly installed by an application, nor will the operating system vector an interrupt request to application level privileged software. Hence, the services of the system agent provide the only way to install device manipulating and interrupt handling software. Further, all such software needs to execute with operating system privilege level. Since this means that the security and stability of operating system is at stake, only tasks with super user privileges are allowed to install and maintain device manipulating and interrupt handling software. The following sub-sections provide an overview of I/O devices, their control by processor and services of this agent to manipulate them.

7.3.1 I/O Devices

An I/O device is a hardware sub-system of a controller that can perform input and output operations. Examples of device sub-systems include input and output interfacing modules with sensors and actuators, serial communication devices, network interface cards and special purpose modules such as motion control and vision processing modules. Devices are usually available as self-contained modules that can be plugged onto a slot in the chassis of a controller system. These modules communicate with the main processor through a standard system bus. A open system bus architecture such as VME or PCI provides the ability to connect a large and diverse number of devices to the controller system.

Though there will be many hardware components to a device depending on its purpose, the most important component from the operating system point of view is its controller. A device controller is responsible for autonomous execution of the device sub-system under the supervision of main processor. A device is manipulated by main processor through its controller. Sometimes several devices may share same controller. For instance, two serial communication devices may share a single controller. The existence of device controller is transparent to the end user.

The main processor communicates with a device controller in order to transfer data and to set control parameters. For example, such a communication may be a simple command to set the baud rate of a serial communication device or a complex command to initiate execution of a small program (sequence of commands) in the memory by a network interface controller. The controller of a device can be addressed by writing or reading bytes at specific address in an address space. Each of these addresses correspond to a control or data port of the controller. Depending on the processor architecture there are two ways to access the port of a controller: memory mapped I/O address space and I/O mapped I/O address space.

In memory mapped addressing, the ports of a device controller are mapped to a range of addresses in the memory space of a processor. When a program reads from or writes to these addresses the ports of device controller are accessed. Memory mapped I/O has the advantage of simplifying instruction set, since no special I/O instructions are required. Similarly no special protection mechanisms are needed other than normal memory access protection mechanism. However, some amount of memory address space is lost due to the allocation to device controllers. It also complicates the data caching mechanism, since most modern processors/controllers use caching to improve their performance. This problem can be worked around if the processor supports a page level caching disable mechanism.

Some processors provide a relatively small address space dedicated to input and output. This space is known as I/O mapped input/output address space. This address space is not accessible by normal load, store or move instructions used for memory access.

Instead, special in and out instructions are used. Further, in order to enforce protection these instructions can be executed only by programs with sufficient privilege level. A disadvantage of I/O mapped addressing is that it makes access of ports more difficult since usually only a few addressing modes are provided with the I/O instructions. Further, exclusive access to an I/O address range is complex to implement and requires special mechanisms.

Accessing a port of a device controller from a high-level language like C or C++ in memory mapped architectures is relatively easier than accessing a port in an architecture that uses a dedicated I/O address space. In the former case a variable bound to the address of the port can be used to access I/O ports. Assigning to the variable is equivalent to writing to the port, while reading the variable is equivalent to reading the port. In architectures with separate I/O address spaces special inline assembler instructions are needed to access a port. The system agent supports both forms of I/O addressing, subject to privilege checking.

The synchronization between the processor and device controller requires a special mechanism. For instance, the processor initiates an I/O operation by sending a command to the device controller. An example of such operation would be to initiate analog to digital conversion in an I/O interface module. The controller executes the command and completes the operation after some time. The processor has to be notified when the operation is completed in order to resume tasks waiting for this completion. One way to accomplish this is through polling by main processor. However, this is very inefficient and is unsuitable for high performance controllers.

Most modern I/O devices provide an interrupt mechanism to notify the processor of a significant situation. Also, all open architecture system buses provide special interrupt lines to communicate such signals to processor. Upon the occurrence of a significant event, the device controller raises an interrupt signal that is communicated to a priority interrupt controller. The priority interrupt controller prioritizes all pending interrupts and is selectively programmable by the processor to enable/disable specific interrupts. At any point of time, the priority interrupt controller communicates the highest priority interrupt

among the pending enabled interrupt signals, to the processor. The processor receives the interrupt signal if it has enabled interrupt processing.

The external I/O interrupts are treated as special cases of exception by most modern processors and the interrupt handling mechanism is similar to exception handling. Most processors support up to 256 exceptions and interrupts. It is the responsibility of operating system to initialize a special data structure of interrupt table containing the pointers to individual interrupt handling routine along with information such as privilege level required to access it. Upon the occurrence of an interrupt, the processor performs several checks to ensure integrity and automatically vectors (jumps) to the interrupt service routine pointed to by the said table. It is the responsibility of interrupt handling software to gather further information about the interrupt and take required actions such as acknowledgment and signaling of service completion.

Though the interrupt mechanism is very efficient in enhancing system performance, it introduces special requirements due to its asynchronous nature. Care must be taken to save execution context and to ensure system integrity through atomic transactions and mutually exclusive access to critical regions and data. The interrupt handling mechanism implemented by system agent reduces the burden on interrupt service routine for such care.

7.3.2 Device Management

The system agent uses an abstract class Device to define virtual interfaces for object of this type. This class defines method interfaces to Probe, Open, Close, Read, Write and Control an I/O device. In order to define a new device driver, one simply has to inherit from this class and override its member functions. Thus polymorphism is put into use when actual method invocation on an object of this type takes place. A new device driver object can be installed into the operating system dynamically by a task with super user privilege through the services of the system agent. The system agent provides Create and Delete primitives to install and remove a device driver object. The Create primitive

takes a reference to the device object and encapsulates it inside the system HW Device object.

The logical object identifier returned by Create primitive refers to the system object. This identifier could be used in the Probe, Open, Close, Read, Write, Control and Delete primitives of system agent. It should be noted that due to the protected nature of DCOS, direct method invocation on a device object is impossible and will result in an exception. The services provided by the system agent are the only way to indirectly invoke a device object services. The services of the system agent raise the privilege level, switch stacks, check memory reference arguments for access violation and redirect the invocation to appropriate object. The result returned by this device object is returned to the original task after lowering privilege level and reverse switching of stacks. All service primitives except for Create and Delete can also be invoked from application privilege level.

The object oriented device management capability provided by system agent is fundamental to online extension of metamorphic control functionality for DCOS. This technique obviates the need to statically link in device driver software as in conventional operating systems and the importance of this can be readily understood from the fact that device driver software typically forms the single largest chunk of code in any traditional operating system. The ability to dynamically install, remove or reconfigure device driver software provides the capability to plug-and-play new hardware online, an important requirement for metamorphic control.

Further, the flexible services of system agent and the dynamic loading/linking capability for application software libraries (described in section 7.12) can be used to add completely new functionality to DCOS online. An example of this would to add a disk drive and a file system to DCOS. In fact multiple types of file systems can be implemented on the same drive using multiple device objects. Careful design of device driver software is necessary to provide transparent fault masking capability in operating system. Most of the fault tolerance capability for metamorphic control system can be implemented through a combination of fault monitoring agents with super user privileges and device driver objects with fault detection/correction capabilities. This approach has been chosen for DCOS

because it is not only impossible to foresee and implement tolerance for all types of faults, but would also result in a huge operating system.

7.3.3 Interrupt Management

Any real-time operating system must provide a mechanism for quick response to externally generated interrupts to satisfy the critical time constraints of the control application. The system agent permits quick interrupt response times by providing deterministic worst case interrupt service latency and critical ability to alter task execution. This allows for an executing task to be preempted upon exit from interrupt service routine. The system agent provides an InstallISR primitive to install a interrupt service routine to a hardware interrupt vector. This service is considered privileged and can be used only by a task with super user privileges.

When an interrupt occurs, the processor will automatically vector to the interrupt handling wrapper of system agent. The system agent saves and restores all registers which are not preserved by the normal procedure calling convention for the target processor and invokes the installed interrupt handling procedure along with the information about interrupt vector number. The invoked interrupt service routine is responsible for processing the interrupt, clearing the interrupt if necessary, invoking appropriate device object(s) methods and any device specific manipulation. The interrupt service routine may complete all the processing requirements itself, if such requirements are small or can reschedule a server task to complete the processing by sending a signal, event or message to it.

The system agent guarantees that proper task scheduling and dispatching are performed at the conclusion of an interrupt service. A system call made by an interrupt handling procedure may have readied a task of higher priority than the interrupted task. Therefore, when the interrupt service completes, the postponed dispatch processing must be performed. Interrupts are nested whenever an interrupt occurs and interrupts are enabled, during the execution of another interrupt service. System agent supports efficient interrupt nesting by allowing the nested interrupt services to terminate without performing

any dispatch processing. Only when the outermost interrupt service terminates will the postponed dispatching occur.

Unrelated low priority interrupts can cause the effect of cycle stealing in a high priority task, thereby causing it to miss critical deadlines. A protection mechanism is therefore required by application tasks to avoid unwanted interruptions. The system agent provides such a protection mechanism to application tasks through a SetInterruptLevel primitive. An application task can specify an interrupt level with this primitive and only interrupts with priority level higher than the set level are allowed to interrupt execution. The operating system maintains a tasks interrupt level whenever the task is executing by restoring interrupt level across context switches.

Interrupt priority level 0 has the highest priority and numerically higher priorities are at lower priority levels. Priority level 0 is assigned to timer interrupts and these interrupts cannot be masked. All other priorities are masked according to application task requirements. This protection mechanism also introduces potential problems by opening the door for abuse. For instance, a lower priority task may block an interrupt meant for higher priority task. To avoid this scenario, the system agent provides an overloaded SetInterruptLevel primitive that can be used only by a task with super user privilege. With this primitive, super user can specify the legal limit of interrupt level for a particular task priority. When lower priority task requests for a high interrupt mask level that is not legal for its own priority level, the operating system will choose the legal maximum. This mechanism protects the interrupt masking facility of operating system from abuse.

7.4 The Scheduler Agent

The concept of scheduling in real-time systems dictates the ability to provide timely response to critical external events. The scheduler agent provides this capability to the real time distributed controller operating system. It is the responsibility of the scheduler agent to allocate processor time to the highest priority task among various tasks competing for attention. The scheduler agent provides a variety of static and dynamic priority based preemptive scheduling mechanisms for this purpose. The flexible

mechanisms provided by the scheduler agent can be used for scheduling hard, soft and non real time tasks of periodic, sporadic or aperiodic nature.

A periodic task is one which must be executed at regular intervals of time. Periodic tasks can be characterized by the length of their period and execution time. The interval between successive iterations of the task is referred to as its period and the deadline of a periodic task is usually the same as its period. The execution time of a periodic task will be less than its period and the ratio of execution time to period determines the processor utilization by this task. Periodic tasks are typically of a hard or soft real time nature. As noted earlier, hard real time tasks cannot afford to miss their deadline since it will result in catastrophe, while soft real time tasks can afford to do so at the expense of quality.

A sporadic task occurs at irregular intervals of time but has a deadline before which it must be completed. Sporadic tasks typically execute in response to external events and usually have a minimum delay between the occurrence of same event. Due to the dynamic nature of sporadic tasks it is generally not possible to predict processor utilization. Sporadic tasks are also of hard or soft real time nature. In contrast, an aperiodic task executes continuously subject to processor availability and not in response to any particular event though it may make use of event information. It does not have any deadlines to meet and is of non-real time nature.

7.4.1 Scheduling Mechanisms

The scheduler agent internally implements two different scheduling mechanisms. The first one is a static priority based preemptive scheduling mechanism meant for non-real time and unit level statically analyzable real time tasks. The second is a combined static and dynamic priority based scheduling mechanism meant for dynamic deadline driven hard and soft real time tasks. The existence of these two mechanisms is transparent to application tasks. The scheduler agent automatically uses one of these mechanisms depending on task context i.e. when a task is within a dynamic deadline based real time scope the dynamic scheduling mechanism is used, while it is outside the static scheduling mechanism is used.

Among these two scheduling mechanisms, the tasks requiring dynamic scheduling are given higher priority over others. In other words, tasks requiring static priority based scheduling will be considered for execution only when there are no tasks requiring dynamic scheduling. The reason being that deadline driven real time tasks are considered more important than others. The rationale for using two different mechanisms is that only tasks with complex real time requirements should have to pay the additional cost incurred due to dynamic scheduling. Though the dynamic scheduling mechanism can be used for static scheduling as well, it will lead to inefficient processor utilization due to higher costs.

The static priority based scheduling mechanism can be used to provide three types of scheduling: round robin scheduling, time shared scheduling and rate monotonic scheduling. In the manual round robin scheduling, a cooperative scheduling policy involving voluntary relinquishing of processor by the executing task is used. The scheduler agent provides a Yield primitive to voluntarily relinquish the processor. The tasks are scheduled in a first come first served manner. In the time shared scheduling, all the tasks get access to the processor resource for one time slice period at a time in an equitable manner. The quantum value of time slice can be set by a super user task during initialization. Time slicing for tasks can be enabled during task creation and tasks are scheduled in a first come first served manner.

These two scheduling mechanisms can be used in conjunction with priority based preemptive scheduling. In this case, the round robin and time shared scheduling mechanisms are applied only when multiple tasks of equal priority are ready for execution, and the tasks of equal priority are serviced in a first come first served fashion. When a task of higher priority become ready for execution, it preempts present lower priority executing task. These two scheduling mechanisms can be only used with non real time tasks and are commonly found in general purpose commercial operating systems.

As explained earlier, the rate monotonic scheduling (RMS) is an optimal scheduling policy for static priority, unit level, time triggered real time systems comprised of periodic tasks. If the periods and worst case execution times for a set of periodic tasks are known a priori, the RMS can be used to guarantee that critical tasks will always meet

their deadline even under transient overload conditions. To do so, RMS calls for static assignment of priorities based upon their execution period i.e shorter a task's period higher its priority. After assignment, all tasks that meet their first deadline when started together are guaranteed to be schedulable.

Static priority scheduling has an efficient and deterministic O(1) implementation that always take a constant amount of time, irrespective of number of tasks ready for execution. The DCOS provides 32 priority levels for tasks ranging between 0-31 with 0 being the highest priority. As shown in Fig. 7.5, the static scheduling mechanism makes use of a bit pattern to indicate which ready queues corresponding to priorities have tasks ready for execution. The ready queues are simple doubly linked lists that have O(1) cost for insertion and removal at tail and head positions.

In order to identify the highest priority task, the scheduler agent performs a high speed bit search that has O(1) cost for practical purposes and schedules the task at head position of ready queue corresponding to bit index searched, for execution. Hence, all the operations involved in static scheduling are performed in constant time. The term static priority does not mean that a task cannot change its priority during run time. Instead it indicates that the priority is not determined dynamically from deadlines. The scheduler agent provides a service primitive ChangePriority to modify static priority during run time.

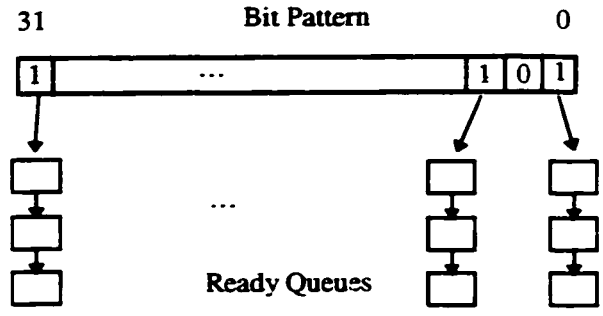


Figure 7.5: Static Priority Scheduling

The dynamic scheduling mechanism of scheduler agent is based on a new scheduling algorithm developed during this dissertation research and is a variation of the Minimum Slack First algorithm. Similar to the Maximum Urgency First algorithm, it uses two priorities, one static and another dynamic. Tasks are first prioritized according to their

static priority and among tasks with identical static priority, dynamic priority is used. The dynamic priority of a task is its latest start time. The latest start time of a task is given by: deadline - execution period. The deadline is calculated as: task arrival time + deadline period. Tasks of equal static priority are resolved according to the ascending order of their latest start times i.e. a task with an earlier latest start time has a higher priority.

The effect of using latest start times is same as Minimum Slack First scheduling. However, the latest start time was chosen because of implementation considerations. A slack is a relative quantity that changes with time as time advances. Hence, a queue with delta slack times needs to be used, but a delta time queue will work if there is only one static priority. On the other hand, latest start time is an absolute quantity and will work for multiple static priority levels. In order to resolve static and dynamic priority efficiently, they are combined into a single value as shown in Fig. 7.6. A 64 bit composite value is encoded by assigning static priority to higher order bits and dynamic priority to lower order bits. This encoded composite value is used to sort the priorities efficiently.

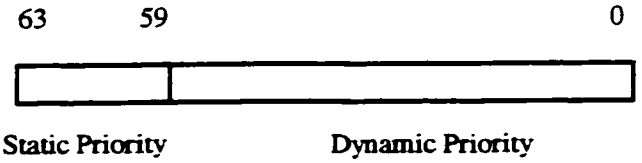


Figure 7.6: Encoded Priority

The encoded priority values might still result in ties. In order to break those ties following rules are used. First an attempt is made to break the tie based on deadline periods. A task with a shorter deadline period is accorded higher priority. The rationale being that shorter deadline periods indicate relative urgency. If the tie still persists, it is broken using shortest processing times. The rationale being that in the case of transient overload, choosing shortest processing time will result in proper completion of first task and proper invocation of scheduling failure handling mechanisms for others, if needed. A system clock with sufficient resolution will prevent a tie at this point. With a coarser clock, the ties are broken based on a first come first served basis.

This dynamic scheduling mechanism is superior to any of the static or dynamic

scheduling mechanisms described earlier. It is better than static scheduling mechanisms because it has a 100% schedulable bound and hence can schedule more tasks than rate monotonic scheduling. At the same time, it ensures critical task set will meet deadline under transient overload, which is unlike other dynamic scheduling algorithms. Further, a pessimistic worst case estimation for task execution period need not be used. Since the execution period can be changed dynamically, an optimistic estimate that reflects the present conditions can be used. Similarly, the deadline period can be changed dynamically to suit current requirements.

The dynamic priority scheduling mechanism though costlier than static priority scheduling, has an efficient $O(\log_2 N)$ implementation, where N is the number of entities in the ready queue. For all practical purposes this implementation can be considered to provide O(1) performance. The implementation makes use of a variation of Binary Heap [Gon91] data structure. As shown in Fig. 7.7, the Binary Heap is a balanced tree that maintains partial ordering of entities in it. The ordering is partial since only the progressive levels are guaranteed to be ordered and there is no ordering within a level. The insertion and removal operations have a $O(\log_2 N)$ cost, since the heap needs to be adjusted after every operation.

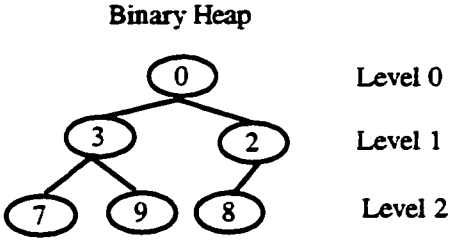


Figure 7.7: Dynamic Priority Scheduling

The Binary Heap represents the prioritized queue of ready tasks and the highest priority task is always at the first location. An important requirement for Binary Heap is that the priorities need to be unique, since the Heap cannot distinguish between identical priorities for ordering such as first come first served. The scheduler agent guarantees unique priorities through tie breaking mechanisms described earlier. The variation from original Binary Heap is due to facts that the scheduler agent generally improves

performance by deferring Heap adjustment after a removal operation and the Heap itself is implemented as a collection of references to objects.

The scheduler agent provides two service primitives, SetDeadline and ClearDeadline to specify real time scope. A task can enter real time scope in two ways: synchronously or asynchronously. In order to enter synchronously, a task uses SetDeadline primitive with set now option as one of the parameters. In this case real time scope starts immediately and a deadline is set. If set later option is used as a parameter, then the task will enter real time scope asynchronously at a later time. The entry happens when a task blocks for first time after using SetDeadline primitive. This blocking may be for anything that will happen or arrive, such as event, signal, message or resource access. On the occurrence of such thing, the task enters real time scope and a deadline is set. The SetDeadline primitive also takes the execution and deadline periods as parameters to facilitate dynamic changes. The ClearDeadline primitive signals the end of real time scope.

7.4.2 Execution Services

The scheduler agent provides services to detect scheduling failure and take appropriate actions. It can detect scheduling failure for tasks that use deadline driven dynamic scheduling, since the requisite information is available only for such tasks. The scheduler agent constantly keeps track of a tasks deadline and if the deadline is crossed before end of real time scope, a scheduling failure is signaled. This is delivered to the task as an exception through normal exception handling mechanisms described later. If the task has installed an exception handler, it will be invoked to service the exception. The exception is ignored otherwise.

The scheduler agent also provides services to profile task execution times dynamically online. Since the intelligent controller is an embedded real time platform with high variance loads, it is difficult to estimate execution times accurately by other means. The scheduler agent profiles and provides timing information dynamically, so that it can be used in calculations and scheduling. It provides two primitives, StartProfiling and StopProfiling, for this purpose. It keeps track of actual time allocated to a task between

calls to these primitives. This time excludes periods when a task was swapped out of context. The primitive GetTimingInfo provides access to timing information dynamically. Such information includes maximum, average and minimum execution times.

7.5 The Timer Agent

The timer agent maintains the system clock, enables time slicing and time-out mechanisms, and provides services to manage a number of virtual software timer objects. It uses the programmable hardware timer through an interface object, to provide periodic timer interrupts. Every interrupt corresponds to a clock tick and is delivered to timer agent by handling procedure for timer interrupt. The value of clock tick is a variable initialized by super user and is equal to an integral number of nanoseconds. The system clock is maintained with nanosecond resolution that is updated every clock tick, hence the granularity may not be same as resolution.

The system clock maintains absolute time and is valid for approximately 18 years from the time it was started before an overflow. As shown in Fig. 7.8, the system clock is maintained as two 64 bit unsigned integers, base and increment. Base number is used initially to store the number of nanoseconds from the start of a nearest base leap year, till the time system was started. The increment stores the number of nanoseconds elapsed since the start of system and is updated every clock tick. The summation of these two quantities reflects the system clock and is used to derive time of day.

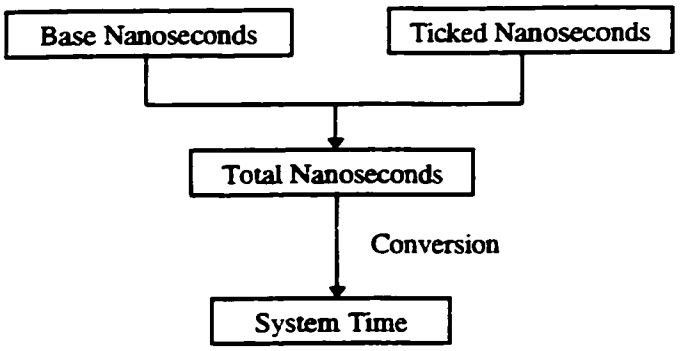


Figure 7.8: System Clock

The time of day is calculated into Gregorian calendar format, by making suitable

adjustments for leap years and century leap years. The timer agent provides two service primitives, GetSystemClock and SetSystemClock to access time of day in Gregorian format. The latter primitive is considered a privileged service and only tasks with super user privileges can change system clock. When the system clock is changed to correct drift from global clock, only the base number of system clock is modified in appropriate way. The new base number is then used in subsequent calculations for time of day to reflect updated time.

The timer agent provides service primitives Create and Delete to instantiation and removal of software timer objects. Once created a timer object may be used to set either interval timer alarm or time of day alarm or periodic alarm. The timer agent provides service primitives SetWatchDogAlarm, SetDayClockAlarm and SetPeriodicAlarm for this purpose. The alarm notification is available either in the form of synchronous event flag or asynchronous signal delivery (described in next section). The notification mechanism may be specified when the alarm is set. The service primitive CancelAlarm may be used to reset a set alarm timer object.

All time out values for timer objects, except for time of day alarm and for service primitives by other agents are specified in nanoseconds using a 64 bit unsigned integer. The time out queue is implemented using an efficient time wheel data structure that is a variation of original schemes proposed in [Varg87]. Instead of using a single delta time out list that is used in most operating systems and provides O(N) worst case performance, a time wheel data structure as shown in Fig. 7.9, is used to provide O(1) average case performance. Though the theoretical worst case performance is $O(N_i)$, where N_i is the number of elements in the i^{th} queue, the performance is O(1) for all practical purposes.

The time wheel structure is comprised of l number of queues, where l is a prime number that is greater than or equal to the estimated number of time outs active at same time. A time out value is subtracted with base nanoseconds and divided by the quantum clock tick, to obtain the incremental tick t at which the timer should expire. The timer is then inserted into the queue that is t modulus l away from first queue. All the queues have their elements sorted according to ascending order of expiry time. The first queue of time

wheel is the queue that is processed when first clock tick occurs.

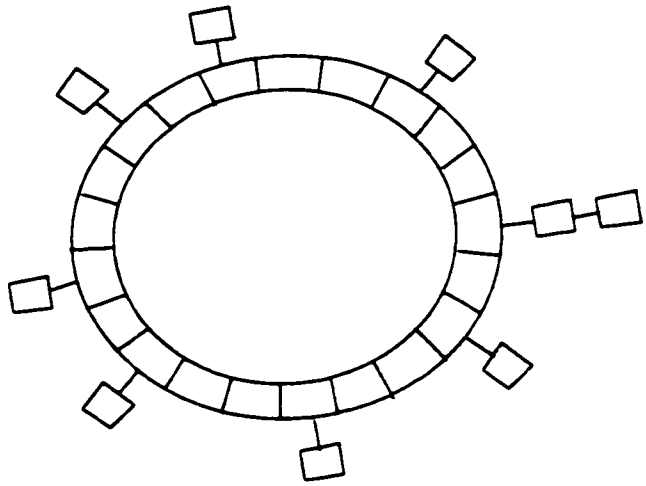


Figure 7.9: Time Wheel Structure

Subsequent clock ticks advance and process the next queue on the wheel. Processing a queue involves checking whether any time outs on this queue has expired, if so removing them and notifying the owner task. Processing a queue is usually a deterministic O(1) operation with sufficiently large time wheel and finer granularity clock ticks, since only rarely will two time outs expire simultaneously.

7.6 The Task Agent

The task agent provides services to manage user level tasks. A system level data structure known as the task object is used to manage associated task. A task is an independent thread of execution control, which can compete on its own for system resources. As explained earlier, a task may be a separate heavy weight process or as a light weight thread within a process. The task agent provides a Create primitive to instantiate a task. A number of task attributes such as priority, stack size, time slicing, floating point, process and visibility, may be specified with this primitive. The priority attribute specifies the initial static priority of the task and may in the range of 0-31 inclusive, with 0 being the highest priority. As noted earlier, multiple tasks can have same priority.

The stack size attribute specifies length of memory region to be set aside for stack. The time slicing attribute specifies whether time slicing should enabled among tasks having

identical priority. The floating point attribute specifies if this task will make use of any hardware floating point unit and enables saving and restoring hardware floating point context during a task switch. This attribute is not applicable to floating point software emulation. It enables efficient context switches by saving and restoring hardware floating point registers only for tasks that need them. The process attribute specifies if the task is to be created as a separate process. The visibility attribute was described before.

As shown in Fig. 7.10, a task is always in one of the following states: non-existent, spawning, ready, executing blocked, suspended and terminating. The transition among these states is also shown in Fig. 7.10. A new task in the process of creation is in a spawning state. Upon completion of this process, it makes a transition to ready state, where it is waiting to be scheduled for execution. Upon scheduled for execution it enters executing state. From this state it may be blocked or suspended from execution depending on the mechanism used to stop execution. It will be readied for execution again based on occurrence or non-occurrence of a requested event.

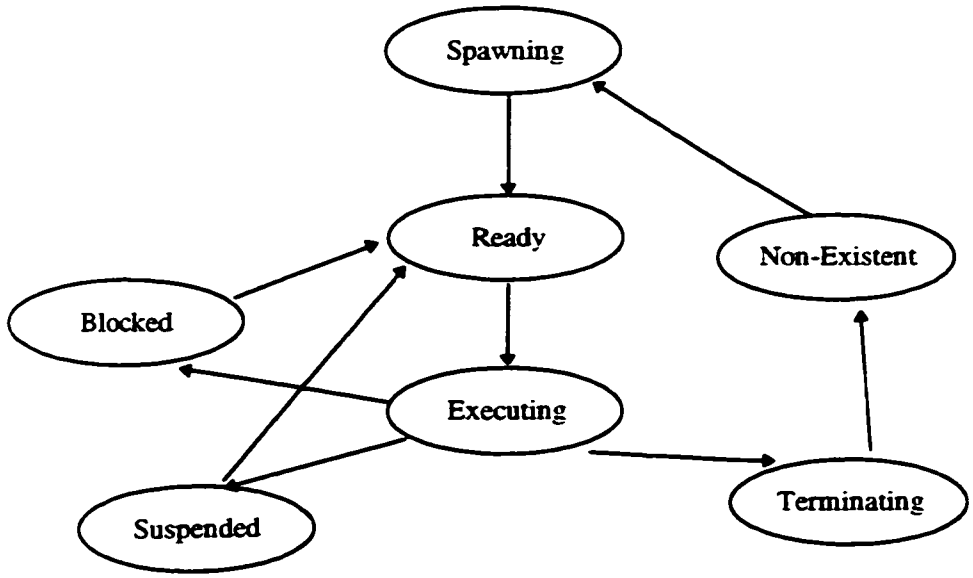


Figure 7.10: Task State Transitions

A task moves into the terminating state when task deletion process is initiated and moves into non-existent on its completion. The task agent provides Delete and KillTask primitives to initiate deletion process. The Delete primitive is a suicidal service that is

requested by same task. The KillTask primitive is a more general primitive that may be used by one task to remove another task. The second task may not even be in executing state i.e. they may be blocked or suspended from execution. The KillTask primitive may be used by a task to kill another task that is a thread within the same process. It may also be used by a super user task to kill another task in a different process.

7.6.1 Task Services

The task agent provides a number of services for user level tasks. Such services include changing priority, simple time delays, event flags, signals and exception handling. The task agent provides a primitive ChangePriority to change the static execution priority of a task dynamically. It provides two primitives SleepFor and SleepTill for simple time delays. The first one uses an interval time out period for which the task will be blocked from execution, while the second one is a time of day variation of same service. These two services use default watch dog timer associated with a task and hence do not require a timer object.

An event flag is used by a task or an interrupt service routine to inform another task of the occurrence of a significant situation. Thirty-two event flags are associated with each task. A collection of one or more event flags is referred to as an event set. Events are independent of one another and are not queued. In other words, if an event is posted more than once before being received, the second and subsequent posting operations have no effect. The task agent provides a primitive PostEvent to set an event flag of a task anywhere in the distributed system. It provides the primitive ReceiveEvent to wait for the occurrence of an event set. A task can wait for the occurrence of any individual event in an event set or can wait for the occurrence of all events in the set. Further it may poll for the occurrence of an event set or specify a time-out condition.

The task agent allows a task to optionally define an asynchronous signal processing routine through the SetSignalHandler primitive. A signal is to a task what an interrupt is to the operating system. When the processor is interrupted, the execution of an application is also interrupted and an interrupt handling procedure is given control.

Similarly, when a signal is sent to a task, its execution path will be "interrupted" by the signal handling procedure and hence signals are also known as software interrupts. Sending a signal to a task has no effect on the receiving task's current state except when it is in a suspended state. In the latter case, the task will be readied for execution.

Thirty two signals are associated with each task and all of them are user defined software interrupts. Similar to event flags, signals are not queued. Hence, multiple signals of same type before first one can be delivered have no effect. The task agent provides a service primitive SendSignal to send a signal to another task anywhere in the distributed system. To deliver a signal the operating system constructs a simulated interrupt context on the stack and invokes signal handling procedure with signal number as argument, if one has been installed. Otherwise the signal is left pending till a handler is installed.

Signal delivery is not nested and multiple signals are delivered sequentially i.e. one signal will not interrupt while another one is being delivered. The priority of signals are predefined with signal 0 having highest priority. However, since signals are not nested, they are also non-preemptive i.e. a higher priority signal arriving when a lower priority one is being delivered will have to wait till the lower priority delivery is complete. Signals can be masked by using the SetSignalMask primitive. A signal set specifying the mask can be installed dynamically by the task or from the signal handling routine. The masked signals are left pending till their mask is removed.

The task agent allows every task to optionally establish an exception handling routine through the SetExceptionHandler primitive. The exception delivery mechanism is used by the operating system to signal exceptions to a task. The exceptions are classified into timing exceptions, synchronous exceptions, asynchronous exceptions and fatal exceptions. The timing exception is the scheduling failure exception discussed before. The synchronous exception include the ones that is caused by invalid instruction operands such as divide by zero and floating point exceptions.

The asynchronous exceptions include the ones caused by system hardware such as bus error. The fatal exceptions include the ones caused by irrecoverable errors such as segment access violation and certain double exceptions. The exception delivery

mechanism is similar to signal delivery mechanism, but has higher priority than signals. Exception may be delivered while signal handling is in progress. Some exceptions such as timing exception are ignored if a exception handler is not installed, while others result in termination of task. An exception handler may recover from certain exceptions such as floating point exceptions by taking suitable actions.

7.7 The Buffer Pool Memory Agent

The buffer pool memory agent provides a high level memory management mechanism for user tasks. Most of the dynamic memory requirements in high level programming languages such as C/C++ are for fixed size data structures. A buffer pool memory offers a high performance and low overhead memory management scheme for such requirements than the regular variable sized memory block management scheme. A pool is a physically contiguous memory area divided into fixed-size buffers that can be dynamically allocated and deallocated. The size of pool memory area and buffers are user defined with the restriction that minimum buffer size should atleast be 4 bytes long.

The buffer pool memory agent provides Create and Delete primitives to instantiate and delete pool objects. The size of pool area and buffer can be specified during creation. It provides Allocate and Return primitives to obtain and release buffers from pool objects. These primitives fail with an error code if they detect inconsistency in a pool object and deleting the object is the only way out. Inconsistency may be caused by an erroneous application overwriting memory regions within process boundaries. The buffer pool memory agent also provides a GetStatistics primitive to access usage statistics for a pool object.

7.8 The Segmented Heap Memory Agent

The segmented heap memory agent provides an alternative high level memory management mechanism that is flexible but costlier than buffer pool memory management. A heap is a physically contiguous memory space from which variable-sized segments are dynamically allocated and deallocated. The size of heap memory area is user defined and

there is an overhead of 8 bytes per segment to maintain segment information. The minimum segment size allocated is restricted to 8 bytes and segments are always data aligned on an eight byte boundary.

The memory allocation requests from a heap object by a user task are processed using first-fit algorithm [Tan87]. In this algorithm, the available memory regions are maintained as a list in arbitrary order and the first region that is larger than or equal to requested size is allocated. If the allocated region has a larger size than requested, then it is fragmented. Upon return to the Heap, a free segment is coalesced with its neighbors on both sides, if any or both of them are free, to produce the largest possible unused region. This algorithm provides quick and satisfactory response so long as the heap is evenly fragmented. Hence it is advisable not to club widely varying segment sizes into same heap.

The segmented heap memory agent provides Create and Delete primitives to instantiate and delete heap objects. The size of heap area can be specified during creation. It provides Allocate and Return primitives to obtain and release segments from heap objects. These primitives fail with an error code if they detect inconsistency in a heap object and deleting the object is the only way out. Inconsistency may be caused by an erroneous application overwriting memory regions within process boundaries. The segmented heap memory agent also provides a GetStatistics primitive to access usage statistics for a heap object.

7.9 The Message Port Agent

The message port agent provides highly flexible communication and synchronization capabilities among tasks through port objects. As shown in Fig. 7.11, it provides a N-to-1 type asynchronous client-server model of communication i.e. N tasks can send messages from anywhere in the distributed system to same port that will be received by one task. Synchronous communication model can be readily realized with ports on either end and there is no limit on number of ports a task can own. A message has a fixed length to ensure real time performance and has a length of sixteen bytes to store user defined information. Messages can be posted to ports using FIFO, LIFO or priority

order. 32 priority levels ranging from 0 to 31 inclusive are available with 0 being the highest priority.

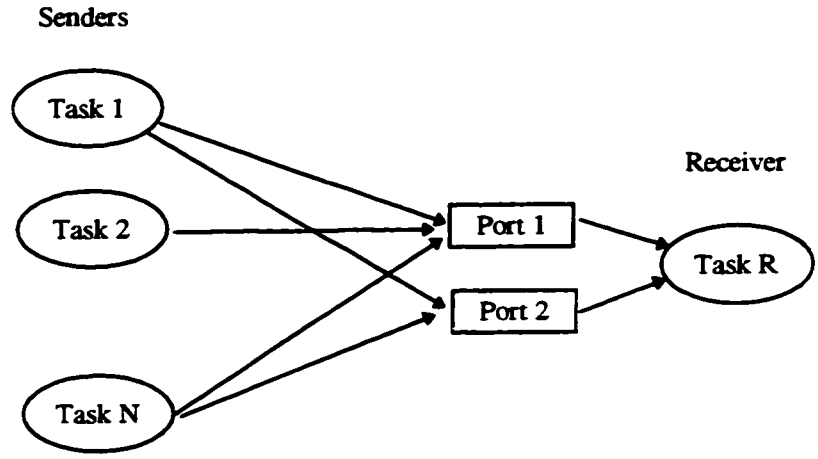


Figure 7.11: Message Ports

The message port agent provides Create and Delete primitives to instantiate and delete port objects. FIFO ports can be specified during creation, otherwise priority ports are created by default. It provides Send primitive to post a message on a port according to specified priority. A negative priority would mean that the message is to be queued in LIFO order at specified priority. Positive priority has no effect on a FIFO port, while any negative priority would mean message is to be queued in LIFO order. There is no limit on number of messages that may be queued in a port. Messages can be received by a task using Receive primitive in the queued order one at a time. If no messages are available, a task may poll or wait with a time out condition for receiving new ones.

7.10 The Distributed Shared Memory Agent

The distributed shared memory agent provides an all software shared memory mechanism with strictly consistent multiple reader-single writer model and 16 bytes fixed page size. Despite the name, it is actually a mechanism for publisher-subscriber model of communication that is crucial for distributed real time systems. As shown in Fig. 7.12, a task can create a publisher object with desired visibility and write onto it from time to time. Depending on visibility this data is published over the network. Another task can

read the published data by opening a subscription to it.

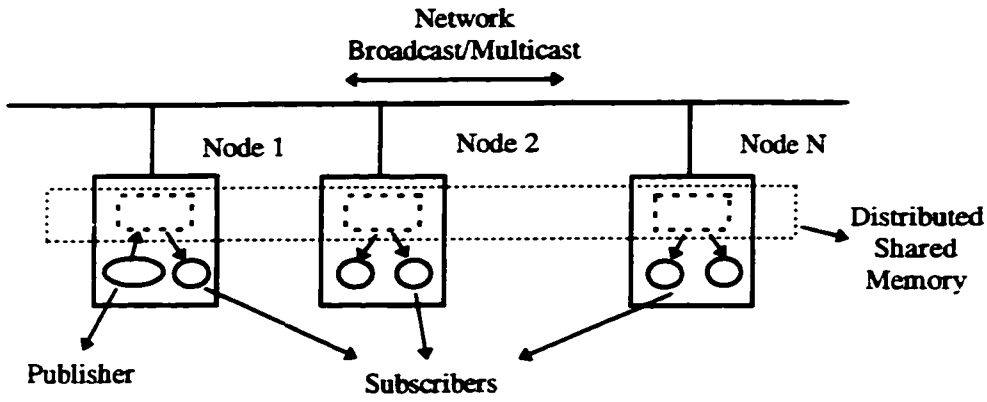


Figure 7.12: Distributed Shared Memory

The distributed shared memory agent provides Create and Delete primitives to instantiate and delete publisher objects. It provides the Write primitive for task owning this object to publish data. It provides Open and Close primitives to create and delete a subscriber object for subscribing and unsubscribing to receive data. It should be noted that a task cannot receive data by circumventing subscription. It also provides the Read primitive to receive data by polling or by waiting with a time out. The Read primitive also takes a data identifier as parameter and this number is used to identify whether the current data is newer than one requested. Similarly when data is read successfully, the associated data identifier is returned.

7.11 The Semaphore Agent

The semaphore agent utilizes standard Dijkstra [Dij68] semaphores to provide synchronization and mutual exclusion capabilities. It supports both binary and counting semaphores for controlling access to local and remote resources in a location transparent manner. A binary semaphore can be used to control access to a single resource. In particular, it can be used to enforce mutual exclusion for a critical section in user code (Mutex). In this instance, the semaphore would be created with an initial count of one to indicate that no task is executing the critical section of code. Upon entry to the critical section, a task acquires the semaphore to prevent other tasks from entering the critical

section. Upon exit from the critical section, the task releases the semaphore to allow another task to execute the critical section.

A counting semaphore can be used to control access to a pool of two or more resources that may be physically distributed in the system. For example, access to three identical devices could be administered by a counting semaphore created with an initial count of three. When a task requires access to one of the devices, it tries to acquire a semaphore thereby acquiring access to a device. If a device is not currently available, the task can wait for a device to become available or return immediately to poll again at a later time (Spin Lock). When the task has completed the operation with the device, it releases the semaphore to allow other tasks access to the device.

Deadlock can occur when a task holding a binary semaphore attempts to acquire that same semaphore and blocks as result. The semaphore agent prevents deadlocks by returning with an error code on detection of such situation. Priority inversion is a form of indefinite postponement which is common in multitasking, preemptive real time systems with shared resources. Priority inversion occurs when a high priority task requests access to a shared resource which is currently allocated to a low priority task. The high priority task must block until the low priority task releases the resource.

This problem is exacerbated when the low priority task is prevented from executing by one or more medium priority tasks. Because the low priority task is not executing, it cannot complete its interaction with the resource and release that resource. The high priority task is effectively prevented from executing by lower priority tasks. The semaphore agent addresses this problem through the optional priority inheritance algorithm. Priority inheritance is an algorithm that calls for the lower priority task holding a resource to have its priority increased to that of the highest priority task blocked waiting for that resource.

Each time a task blocks attempting to obtain the resource, the task holding the resource may have its priority increased. The implementation of the priority inheritance algorithm takes into account the scenario in which a task holds more than one semaphore. The holding task will execute at the priority of the highest priority task blocked waiting

for any of the semaphores the task holds. Only when the task releases all semaphores it holds will its priority be restored to the normal value. Priority inheritance is available as an option for local binary semaphores. It is not supported for remote semaphores due to the costly communication overhead.

The semaphore agent provides Create and Delete primitives to instantiate and delete semaphore objects. The type of semaphore, initial count and optional priority inheritance if applicable may be specified during creation. It provides Acquire and Release primitives to obtain and release resources from/to local or remote semaphore objects. A task can poll or wait for a resource to become available. In the latter case, it can specify a time out condition in nanoseconds. If the resource does not become available within the specified period, the Acquire primitive returns with an error code.

7.12 The Dynamic Linker

When a file containing source code in a high level language such as C/C++ is compiled, the compiler generates an object file as output. An object file is the machine code equivalent of its corresponding source file. Generally, it contains a text segment with machine code equivalent for procedures and a data segment with machine representations of global and static variables, objects, string constants, etc., global symbol definitions and information that enables this module to be relocated. Multiple object files can be combined together into an executable image file through the process of link editing or simply linking.

The link editing activity maps each object module to a region of the run time memory address space, relocates symbol references within a module to new locations, resolves global symbol references across modules, allocates storage for the global data structures and writes the resulting executable image into a file. Object modules to be linked together may be in the form of individual object files or library archives. In the latter case it is the responsibility of a link editor or linker to search through the library archives to ensure all and only required modules are linked in. The format of the resulting executable image file conforms to the specification by the operating system.

Most traditional operating systems support only static linking and program loading. In static linking, the link editing step is carried out only once to produce an executable image file. This image file contains all the information required to create a run time process. The executable file is loaded into memory by a program loader that creates run time memory regions and maps the image regions into memory regions as specified in image. Static linking and loading requires that all global symbols be well defined at link time. Static linking does not allow code to be shared across process boundaries and is not flexible.

Unlike static linking, dynamic linking allows unresolved global symbols to be defined during run time. Most modern operating systems support dynamic linking during program loading and some provide facilities to accomplish dynamic linking at a later stage also. The DCOS provides a kind of online dynamic linking that is different from most operating systems. The dynamic linker allows user to add, remove, replace, or relocate object modules during execution. In other words, application programs could be downloaded on the fly and are allowed to change.

During the lifetime of its execution, a program may have new modules added, old modules removed, or even evolve into a completely different program. For instance, faulty portions of an application program can be replaced on the fly. Hence for a compiled language such as C/C++, the traditional concept that the code of a program cannot change during execution is no longer valid. All processes share same code segment facilitating usage of single copy of machine code. This segment is write protected from all processes for safety considerations.

The dynamic linker maintains an online symbol table of all object modules to link with new ones. It also maintains an explicit reference list of relocations for every symbol. This list is used to restore the referencing locations to their original value, when a symbol is removed online with its object module. A new object module with identical symbol as old one, but with new code can now be linked in dynamically. Thus the dynamic linker facilitates upgrading faulty code online. However, care must be taken to suspend tasks from executing unreferenced locations during this period.

Code sharing requires binary object files containing position independent code. Position independent code locates every symbol indirectly by indexing into an on-line symbol table whose location is held in a special reserved register. This process causes a severe degradation in performance. However, most of the capabilities of position independent code can be simulated through object oriented programming. Hence DCOS supports code sharing through thread safe programming and object oriented mechanisms. The dynamic linker uses the Executable and Linking Format (ELF) [ELF94] for binary object modules.

7.13 The Network Interfaces Manager

The network interfaces manager provides crucial ability for DCOS to function in a distributed multiprocessor environment. The presence of network interfaces is transparent to application tasks, but its services are used internally by all operating system agents. The network interfaces manager implements a routing module and interfaces to multiple communication protocol stacks. The protocol stacks in turn implement protocol related services and interfaces to physical network driver software. It is possible for multiple protocol stacks to share same physical network driver, thereby making it possible to implement multiple protocols over single network.

The network interfaces manager uses a three level address mapping scheme to provide maximum flexibility in terms of multi-homed network interfaces for multiple network types. As shown in Fig. 7.13, the logical address of a node is first mapped onto a network address and then onto a physical address. The logical address is same as the 16 bit node identifier discussed earlier. The network address is a 32 bit address uniquely identifying a protocol stack. The physical address uniquely corresponds to a network interface and has network specific format and length. It should be noted that a logical address may be mapped to multiple network addresses. Similarly a physical address may mapped to multiple network addresses. Hence, the reverse mapping is not as simple as forward mapping.

Figure 7.13: Address Mapping

The network address format has been adapted from Internet Protocol (IP) V4 [IP81]. As shown in Fig. 7.14, it is comprised of three portions, namely, net id, sub-net id and host id. While the net id has a fixed length of 8 bits, sub-net id and host id have variable length depending on sub-net mask length. As in IP, a sub-net mask is used to identify whether the packet is meant for a specific sub-network. Addresses containing the first octet between 128 and 223 inclusive are used as network addresses, while the ones between 224 and 239 inclusive are used as multicast addresses. Address 255.255.255.255 is used as limited broadcast address, while x.255.255.255 is used as net x directed broadcast address.

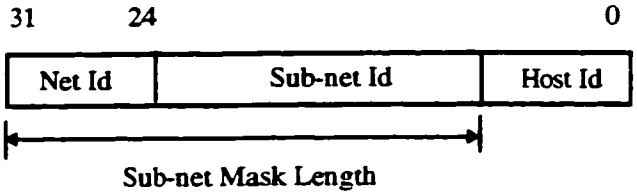


Figure 7.14: Network Address

A high level message is wrapped inside a protocol specific packet by the protocol interface, which in turn is wrapped inside a physical network specific frame by the network driver for communication. The frame is then scheduled for transmission according to message priority. The message priority is mapped onto network priority if available, to resolve global transmission scheduling. The network drivers perform actual transmission of frames by interacting with network hardware or multiprocessor system bus as appropriate. Conversely at the receiving end, the network driver receives a frame and passes the enclosed packet to protocol stack which in turn passes the enclosed message to the network interfaces manager. If the message was meant for local node it is passed onto the appropriate agent for further action.

Fig. 7.15 shows a distributed system comprised of two networks. In order for the

message sent by node 3 to be received on node 1, it has to be routed by the network interfaces manager at node 2. In this case, the network interfaces manager at node 3 forwards the packet to gateway node 2. The one at node 2 identifies that the packet needs to be routed to another network and retransmits it on the second network to node 1. If the system were to be more complex with multiple networks to pass through, a local network interfaces manager routes a message by hopping it to nearest gateway. The routing and address tables can be dynamically constructed by a task with super user privileges through service primitives of network interfaces manager.

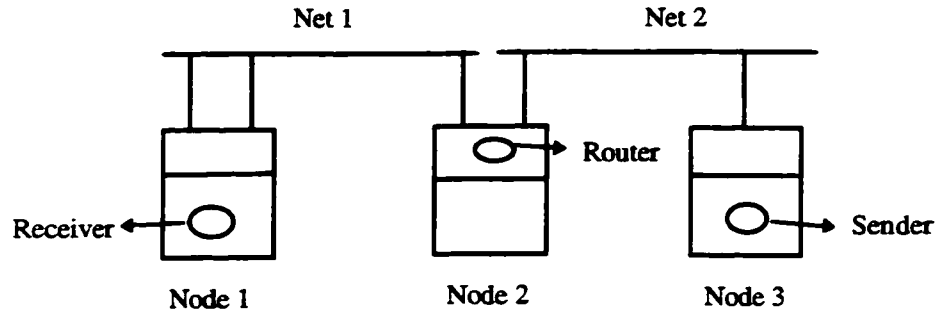


Figure 7.15: Message Routing

Sharing data within a heterogeneous distributed system poses a difficult problem due to the varying data representation schemes used by different processor types. The most pervasive data representation problem is the order of the bytes which compose a data entity i.e. little endian or big endian. Unfortunately, sharing a data structure between big endian and little endian processors requires translation into a common endian format. Other issues include representation of floating point numbers, bit fields, binary coded decimal data, time, date and character strings. In addition, the representation method for negative integers could be one's or two's complement and the floating point precision may be of different word length.

In order to provide maximum flexibility and efficiency, the network interfaces manager does not impose any specific data representation scheme. Instead, system level messages are encoded into network specific schemes, while the application level data is left untouched. This is necessary for data sharing among function blocks, since IEC 1499

requires a specific scheme for encoding data. The actual encoding in this case, can be accomplished with the help of an application level software library. If needed, other schemes for encoding data can also be accommodated simultaneously with suitable libraries.

7.14 The System Engineering Interface Agent

As shown in Fig. 7.16, the system engineering interface agent acts as a remote liaison for its name sake. It is a task or tasks if multi-threaded, with super user privileges that helps the remote commands from any system engineering interface in a distributed system be turned into local actions. It provides services to configure the local components of DCOS from a remote station. It implements simple file transfer protocol to dynamically download program distribution units and other data files into local platform. It also maps the downloaded object modules to a memory regions within the code segment and initiates the dynamic linking process.

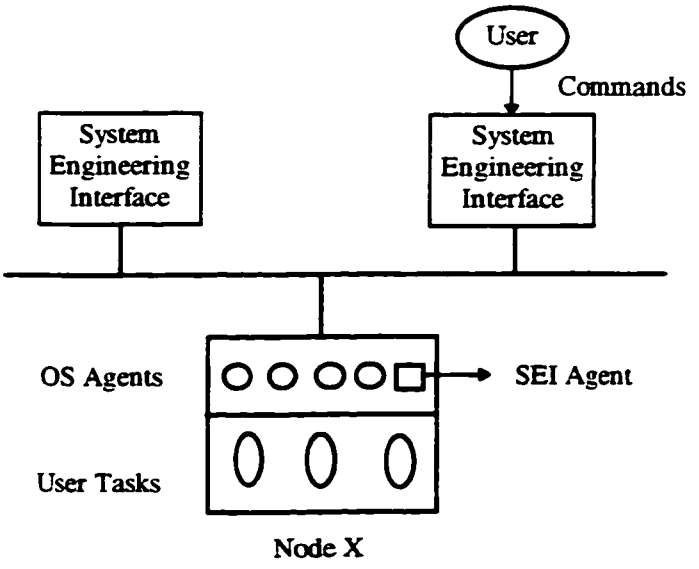


Figure 7.16: System Engineering Interface Agent

It provides services to dynamically unlink previous loaded files and unmap them from code segment to recover memory. It provides services to create and delete tasks that may be threads or processes, and query object identification, file location and symbol

location information. It provides services for local tasks to log messages on any remote system engineering interface. It also provides services for logging real time data into databases distributed among system engineering interfaces. In short, it acts as the primary mechanism to implement manager function block discussed in chapter 4.

7.15 Summary

In this chapter, various components of the DCOS operating system and their implementation were discussed. The important service primitives offered by these components and the algorithms used were also reviewed.

Chapter 8

Application Development and Configuration

8.1 Introduction

The capabilities of a metamorphic control system can only be fully assessed with complex control applications. The ease of modeling, development and configuration of complex applications dictates the practical usefulness of a control system. As described earlier, the metamorphic control architecture facilitates modeling of complex applications using the powerful IEC 1499 function block specification standard. It should be noted, however, that there is no unique way of modeling an application with function blocks. Irrespectively, a model needs to be translated into corresponding application software for execution with desired behavior and real time responses.

Much of the difficulty associated with development of distributed application software is reduced by the location transparent services of DCOS described in preceding chapter. The DCOS has been designed to provide generic and flexible support in implementing event driven distributed control systems. As in modeling, it is possible to implement a single function block model of an application in more than one way. This chapter describes one feasible method for modeling with function blocks and development of distributed application software in C++, using simple illustrations, with implementation through DCOS primitives. A system engineering interface for remote software development, configuration and status monitoring is also described.

8.2 Software Synthesis

As shown in Fig. 8.1, the application software development process involves four major steps. The first step involves modeling of a distributed application with function blocks. The boundaries of function blocks may arise naturally from application characteristics or in some cases defined arbitrarily by system modeler. The second step

involves converting function blocks into source code supported by underlying DCOS services. The code should produce equivalent run time behavior and satisfy real time characteristics of function blocks. In the next step, the source code is cross compiled for target run time environments to produce binary executable modules. In the final step, these modules can be downloaded to target platforms and configured with run time data to create a specific instance of application. It is possible to create multiple instances of an application or parts of it depending on requirements. Software reuse is facilitated both at source code development stage and at configuration stage.

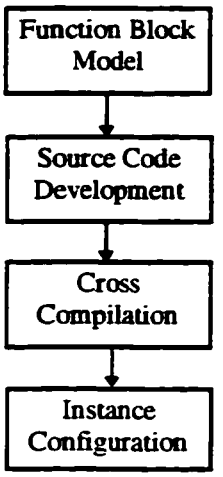


Figure 8.1: Steps in Application Software Development

The application software development process will be illustrated with two simple sample applications, namely, PID application and publisher-subscriber application. While the former illustrates simple function block network with client-server type communication, the latter illustrates fault tolerant two way active redundancy network with publisher-subscriber type of communication and incorporation of symbolic intelligence in the form of fuzzy logic. Though these illustrations do not exemplify development of intelligent reactive behavior based agents and their complex interactions, they do involve the essentials involved in developing distributed agents based real time control applications. Complex reactive behavior based agents and applications can be readily built by scaling these fundamental concepts.

8.2.1 PID Application

As shown in Fig. 8.2, the PID application is modeled by four function blocks in a sequential network. The E_CYCLE function block is a standard IEC 1499 function block to provide periodic timer events. These events activate rest of the network periodically providing behavior similar to traditional sampled data control system. The ADC, PID and DAC function blocks are application specific, and represent analog process variable input, process controller and analog controlled variable output respectively. All four function blocks in network may eventually be configured to a single controller node or assigned two each to two distributed nodes or in any other combination.

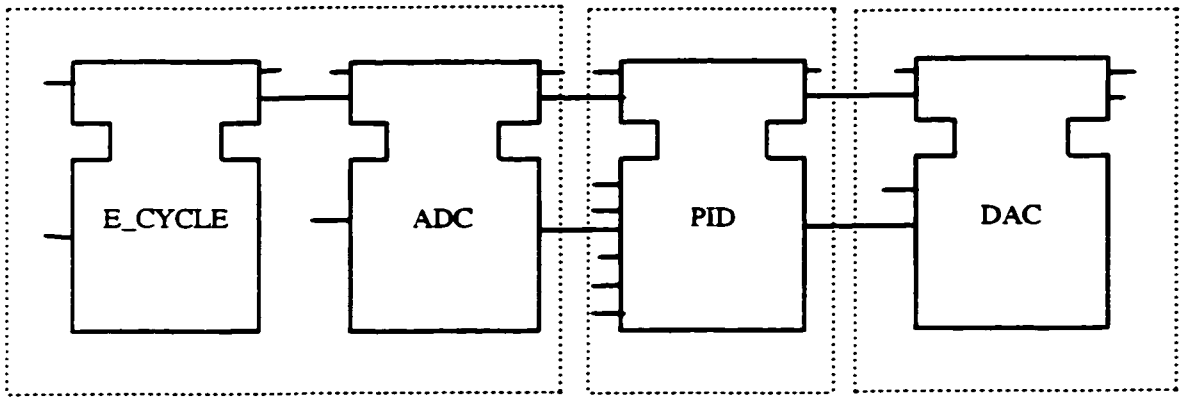


Figure 8.2: PID Application

Figures 8.3, 8.4 and 8.5, show event inputs/outputs, variable inputs/outputs and execution control chart (ECC) details of application specific function blocks. The ADC function block has two event inputs: EI_Init corresponding to initialization command and EI_Sample corresponding to sampling process. The occurrence of EI_Init signifies initialization event with Param as its parameter. This causes ECC to clear transition and move to Init state. This causes initialization algorithm to be executed and on its completion, EO_Init output event is issued. Similarly, EI_Sample input event causes analog sampling process to occur and the converted digital data is mapped to PV output variable with the issuance of EO_Sample output event.

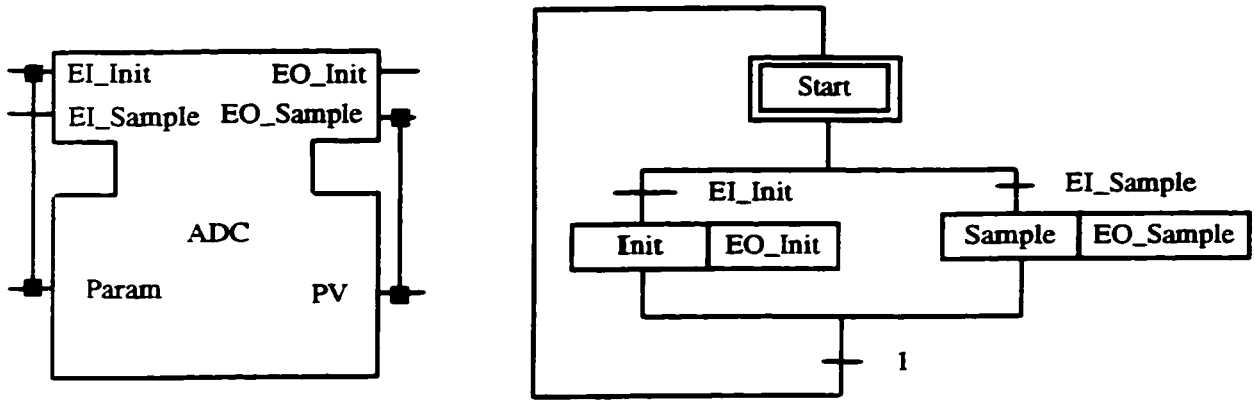


Figure 8.3: ADC Function Block

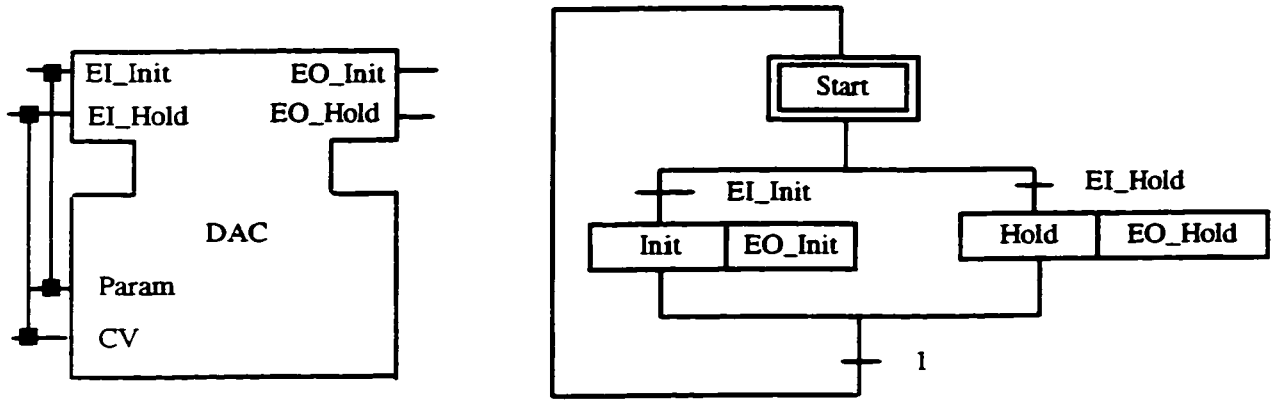


Figure 8.4: DAC Function Block

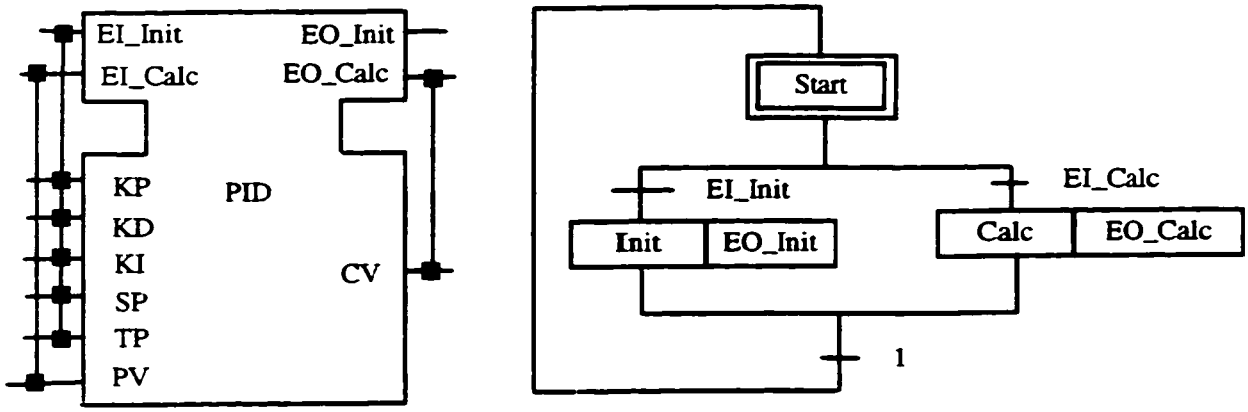


Figure 8.5: PID Function Block

The DAC function block has two event inputs: EI_Init corresponding to initialization command and EI_Hold for holding digital to analog conversion value. The ECC functions similar to ADC function block, while the controlled variable data value for EI_Hold event is passed through CV input variable. The PID function block implements

standard discrete PID equation and the values for proportional, integral and derivative gains, set point and sampling time period are initialized through input variables KP, KI, KD, SP and TP respectively. It also has two input events, EI_Init and EI_Calc for initialization and calculation respectively. The ECC functions in a similar manner as other function blocks, with the input process variable being mapped to PV and ouput controlled variable to CV.

During normal operation of PID application, periodic events from E_CYCLE function block activates analog sampling process. The completion of sampling process activates PID function block for control, with value of converted digital process variable. The completion of PID algorithm triggers digital to analog conversion and hold process, with value of computed controlled variable. Thus traditional sampled data control is accomplished in a event driven distributed control system.

8.2.2 Publisher-Subscriber Application

Fig. 8.6 shows the publisher component of second sample distributed control application. It is composed of three function blocks: E_CYCLE, ADC and Publish. The standard E_CYCLE function block provides periodic timer events to initiate analog process variable sampling. The converted data is broadcast/multicast over network by IEC 1499 standard Publish function block. Two copies of the publisher component are instantiated in two independent nodes by application for two way active redundancy fault tolerance and voting. The voted values will be evaluated by subscriber component of application for accuracy and validity through special logic such as continuity from past data.

Fig. 8.7 shows the subscriber component of application which is composed of five function blocks: 2 Subscribe, E_CYCLE, Fuzzy and DAC. The Subscribe function blocks receive data from publisher component periodically and make the data available for use. The standard E_CYCLE function block provides periodic timer events to Fuzzy function block to read subscribed data and perform computations for control. The E_CYCLE function block has same time periods as publisher components, but provides independent

activation events to Fuzzy function block. This ensures that the application will be operational even if a publisher component fails.

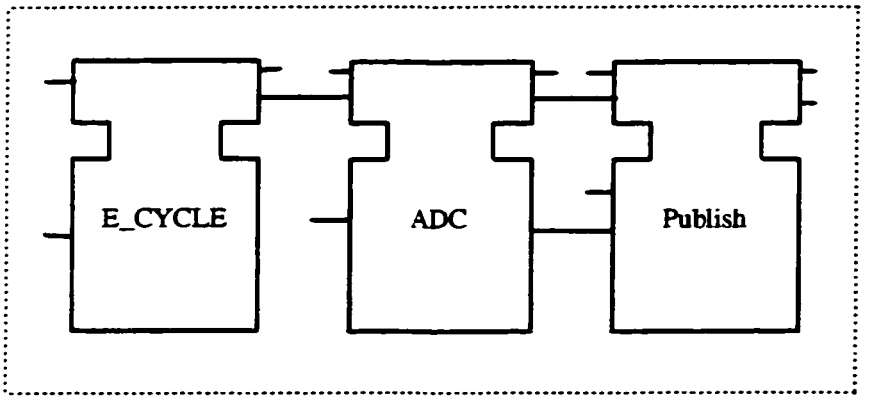


Figure 8.6: Publisher Application Component

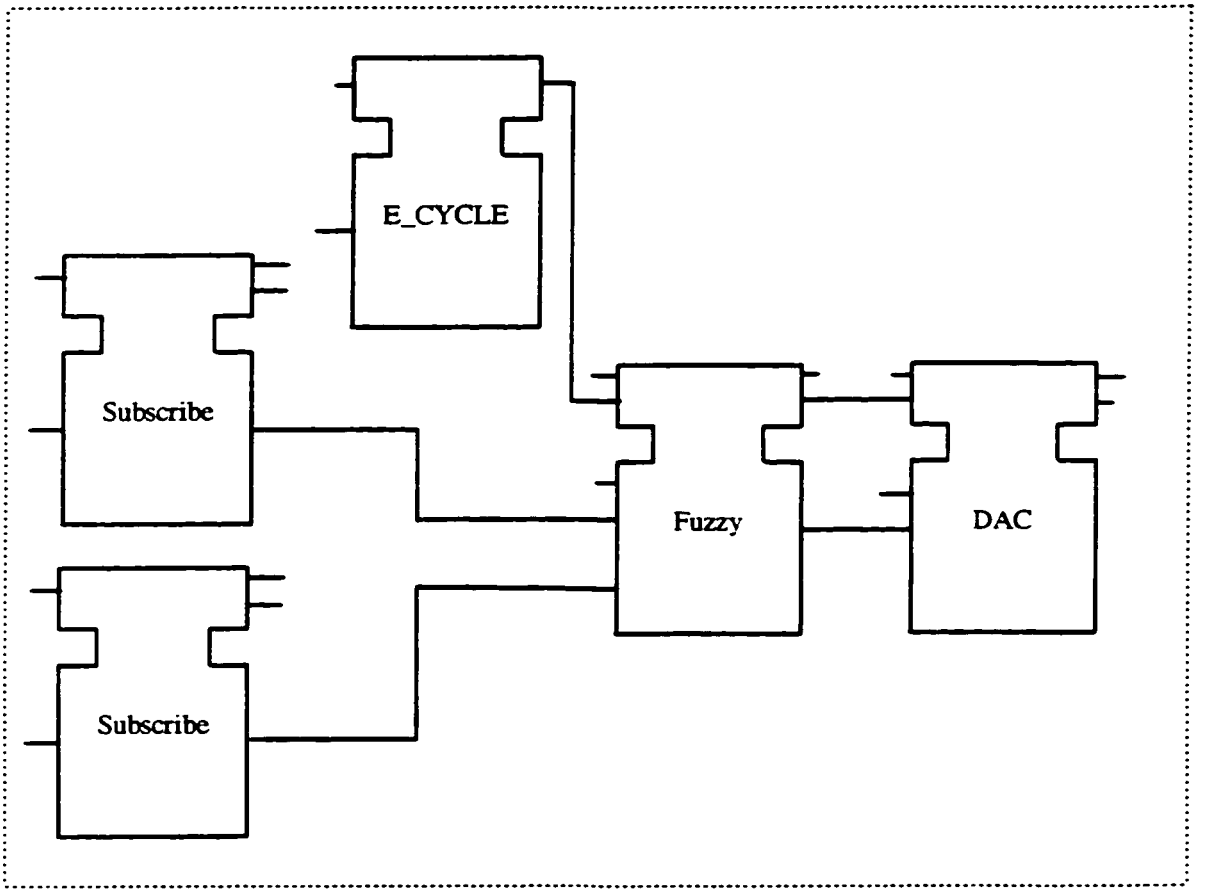


Figure 8.7: Subscriber Application Component

The Fuzzy function block utilizes both subscribed process variable values to arrive at an accurate estimate of process status. The corrective action is computed through fuzzy reasoning and the new controlled variable value is made available to DAC function block for digital to analog conversion and hold. Similar to publisher components, two copies of subscriber component are instantiated in two independent nodes by application for two way active redundancy fault tolerance. It should be noted that publisher and subsriber components are instantiated in separate nodes. As shown in Fig. 8.8, Fuzzy function block has two input events, EI_Init and EI_Main for initialization and normal operation respectively. The behavior of its ECC is similar to that of other function blocks.

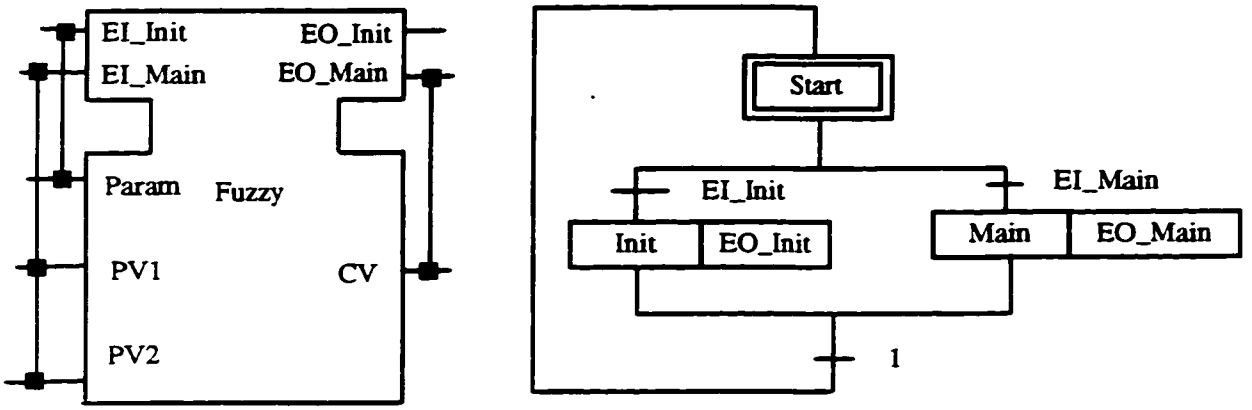


Figure 8.8: Fuzzy Function Block

8.2.3 Code Development

The function blocks described thus far can be readily converted into simple distributed control agents. This is done through developing appropriate C++ classes representing the behavior of these function blocks and instantiating objects of those classes. However, instead of instatiating them as traditional passive objects, they are instantiated as independent active objects (i.e. agents) through user level tasks of DCOS. A local application component is defined within a single DCOS process boundary, with function blocks as independent threads of execution control. The agents communicate with each other through message ports provided by DCOS.

As shown in Fig. 8.9, classes for all function blocks are derived from class BasicFunctionBlock. This parent class defines common interfaces and behaviors such as ECC data structures, ECC execution and communication through message ports. The child classes define function block specific variables, actions corresponding to ECC states and initialize ECC state machine data structure. The event and data communication, ECC execution and state action execution closely simulate the behavior of function blocks in software.

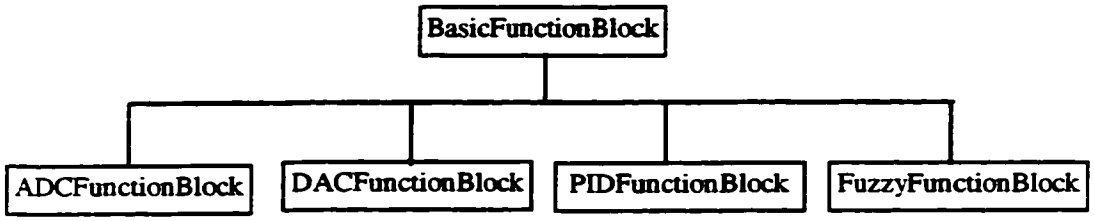


Figure 8.9: Application Class Design

Standard function blocks such as E_CYCLE, Publish and Subscribe are accomplished through suitable DCOS objects and primitives to manipulate them. For example, the functionality of E_CYCLE function block can be obtained by creating a DCOS timer object and manipulating it through the service primitives of DCOS timer agent. Similarly, the functionality of Publish and Subscribe function blocks can be obtained through the creation of DCOS publisher and subscriber objects and manipulating them with DCOS distributed shared memory agent primitives. Further details about application code development with DCOS primitives are described in the DCOS programming manual [Bala97].

In order to achieve software reuse, the agents do not communicate directly with each other. Instead the communication is redirected through call back functions provided by respective application object during initialization. This technique facilitates reuse of the same function block code in multiple applications with different network configurations. The code development process described thus far is sufficient for simple sample applications discussed earlier. For complex applications, additional mechanisms will have

to be used to ensure data consistency. Support for such mechanisms are available from DCOS in the form of mutual exclusion and synchronization services.

8.3 System Engineering Interface

The system engineering interface facilitates remote management of distributed control system using icons. As shown in Fig. 8.10, its functionality can be divided into three areas, namely, programming interface, configuration interface and monitoring interface. The programming interface facilitates remote software development and software reuse. It provides standard libraries for data types, functions and function blocks that are defined by IEC 1131-3 and 1499 standards. It provides support for development of user defined data types and functions. These in turn can be used to develop user defined basic and composite function blocks, and distributed applications. The programming interface also provides cross compilation tools to produce binary object modules for target controller platform.

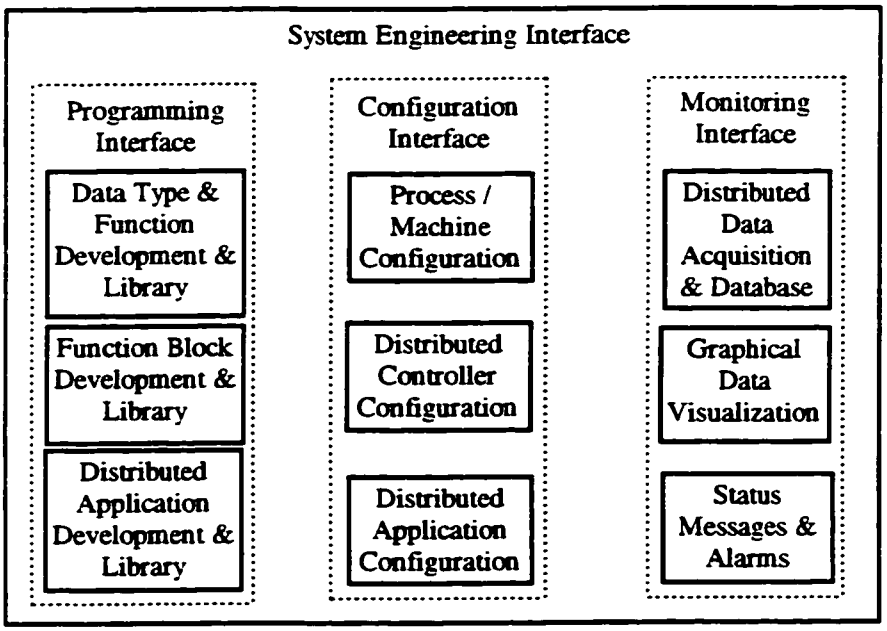


Figure 8.10: Elements of System Engineering Interface

The configuration interface provides facilities to configure process/machine specific information such as distributed controllers involved in controlling them. It also

provides facilities to remotely configure distributed controller resources such as DCOS system objects, networks and addresses. As indicated in Fig. 8.11, applications can be "dragged and dropped" into controllers, at which point binary object modules are dynamically downloaded to remote controllers and linked in. Application copies can be dynamically instantiated and configured with instance specific data. Facilities are also provided to dynamically query information and status, and to unlink and unload binary object modules from remote controller nodes.

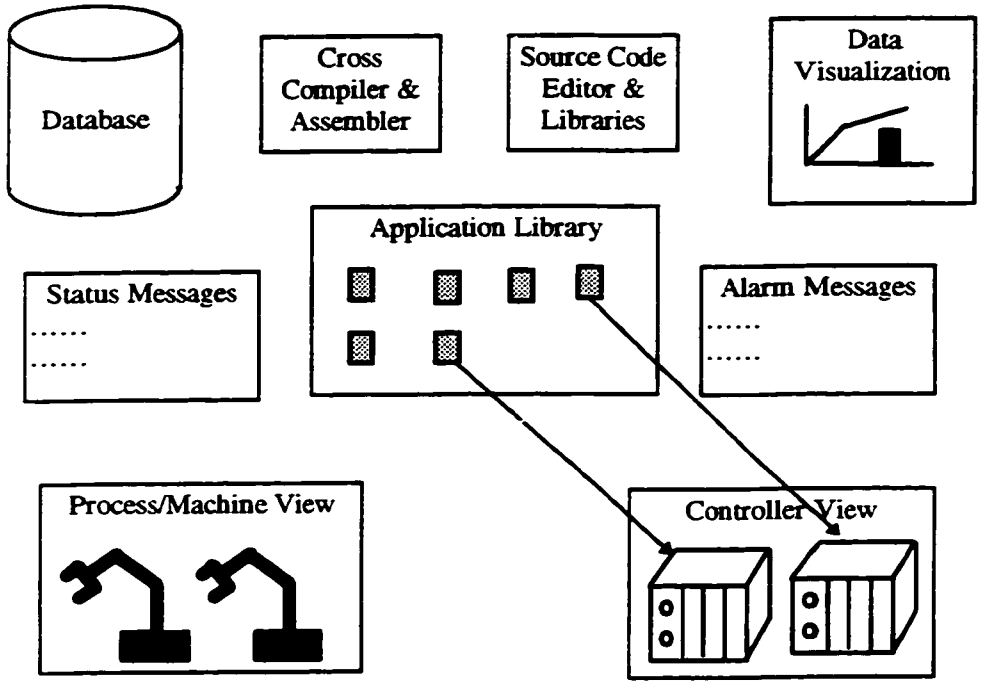


Figure 8.11: Application Configuration

The monitoring interface provides facilities to acquire remote real time data and log them into a local database. It provides facilities to graphically view the acquired data in the form of charts and trends. Remote application tasks can log status messages onto a screen in monitoring interface. Similar facilities are also provided to log alarm messages from remote controller nodes. The system engineering interface was developed in C++ and uses capabilities provided by the system engineering interface agent on distributed controller nodes to accomplish remote management functions.

8.4 Summary

In this chapter, the issues associated with development of distributed application software were discussed. The four major steps involved in software development, namely, modeling, code development, cross compilation and instance configuration, were illustrated with simple examples. A system engineering interface developed for remote management of distributed control system was presented. This system provides support for off-line software development through a programming interface, configuration of process, controller and application parameters through a configuration interface, and data acquisition, visualization, continuous monitoring of system status and alarms through a monitoring interface.

Chapter 9

Implementation and Evaluation

9.1 Introduction

This chapter discusses details pertaining to implementation and evaluation of a prototype metamorphic control system. In the following section, a brief description of controller platforms and networking infrastructure used is presented. This is followed by information pertaining to measurement of performance, methods of gathering timing data and their usefulness. Also discussed are other time critical aspects of DCOS that affect applications design and ultimate throughput. These aspects include determinacy, interrupt latency and context switch times. The performance data for various operating system service primitives are also provided. Finally, some of the tests used in the functionality evaluation of implemented system are presented.

9.2 System Implementation

As Fig. 9.1 shows, the proof of concept system implementation is comprised of a system engineering interface, a uniprocessor and a multiprocessor controller nodes.

The system engineering interface was implemented on a PC running Windows 95 operating system. It makes use of underlying socket based networking services and graphical user interface to implement functionality discussed in preceding chapter. Local database and data visualization capabilities were implemented using commercially available software. Additional capabilities such as remote system management, application software development, cross compilation tools, configuration and dynamic downloading, were also implemented. The PC communicates with controller nodes using an Ethernet based local area network and UDP/IP protocol.

Controller node 1 was based on a commercially available uniprocessor platform running an Intel 80486DX 33 MHz processor. Programmable interval timer hardware was

available in the form of Intel 8254 chip and priority interrupt controller in the form of Intel 8259 chip. The platform used an ISA system bus wherein all system memory accesses were through system bus making it slow. External devices were connected through expansion slots of ISA bus and external interrupts were directed through standard mechanisms of this bus. The local area network accesses were controlled by an Intel 82595 coprocessor.

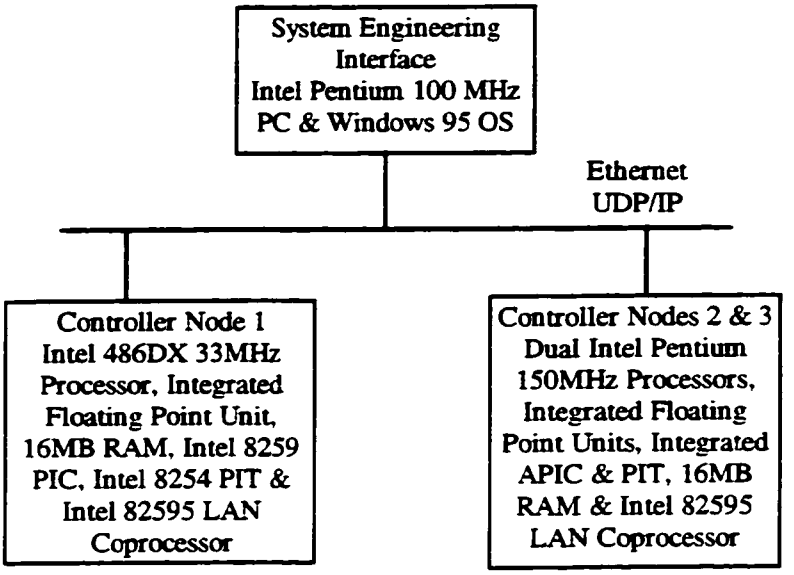


Figure 9.1: System Implementation

Controller nodes 2 and 3 were based on a commercially available multiprocessor platform running dual Pentium 150 MHz processors. This dual processor architecture was based on Intel multiprocessor specification [SMP95] and is shown in Fig. 9.2. The architecture used PCI system bus for local system memory accesses and EISA system bus for expansion slots. The external devices were connected through EISA system bus and external interrupts were redirected from EISA bus to PCI bus. As shown in Fig. 9.2, the interrupts were delivered to Intel 82489DX chip called I/O Advanced Priority Interrupt Controller (APIC).

The I/O APIC bus is connected to a special Interrupt Controller Communications (ICC) bus which in turn is connected to local APICs integrated with processors. The I/O APIC can be programmed to deliver external interrupts to any processor on ICC bus and

local APICs can be programmed to deliver several types of inter-processor interrupts for communication between processors. Both external and inter-processor interrupts are delivered through ICC bus and simultaneous interrupts are arbitrated automatically by ICC bus. The arbitration is based on dynamic priority mechanism that is programmable through local and I/O APICs.

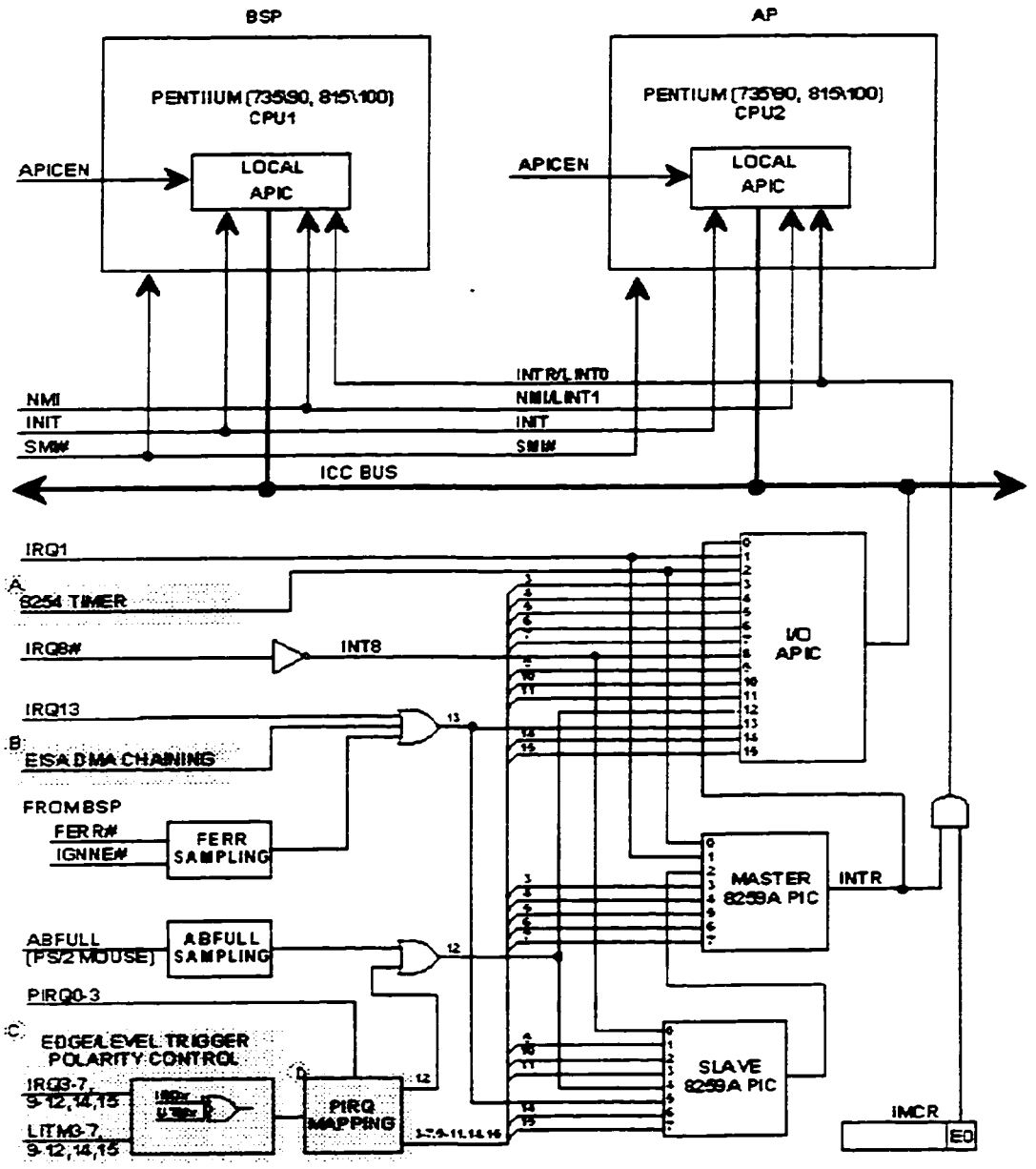


Figure 9.2: Dual Processor Architecture

Though this dual processor architecture was meant for symmetric multiprocessing, the platform was suitably configured to provide asymmetric multiprocessing. This was done since multiprocessor controller platforms typically perform asymmetric processing only. Controller nodes 2 and 3 were assigned different resources for asymmetry. For instance, controller node 2 had direct access to network through an Intel 82595 LAN coprocessor and node 3 through shared memory and inter-processor interrupts, while node 3 can communicate only with node 2 through shared memory and inter-processor interrupts. This meant that node 2 had to act as router for node 3 for external communications. Similarly, external interrupts were assigned asymmetrically among processors.

Both processors made use of an integrated programmable timer to provide periodic interrupts for maintaining system clock. The DCOS was augmented with device drivers software for Intel 82595 LAN coprocessor and communicating with inter-processor interrupt and local APICs. It was also provided with a connection less UDP/IP stack as protocol interface for network interfaces manager. The DCOS messages were transparently mapped to UDP/IP packets and in turn to Ethernet frames by network interfaces manager through protocol interface and was communicated by underlying device driver software.

9.3 Timing Analysis

The evaluation of the implemented metamorphic control system has proven to be a difficult task. This is primarily because of two reasons: most of the metamorphic capabilities are not quantifiable and there are no other similar event driven distributed control systems available for comparison. Metamorphic capabilities such as dynamic online extension/modification of functionality, ease of distributed control application development and reconfiguration of applications are features that cannot be readily quantified. Quantifying such features involve subjective evaluation which depends on need and hence is not unique.

Table 9.1 shows qualitative comparison of major metamorphic control features of DCOS against some of the popular commercial and research real time operating systems, namely, QNX [QNX93], OS-9 [Micro91], VxWorks [WRS94], pSOS [ISI93] and Chorus [Chorus96]. However, this comparison is not a straight forward process, since DCOS is meant for event driven distributed control systems, while the others are meant for time triggered control systems with network connectivity.

Table 9.1 - Qualitative Feature Comparison

Features	DCOS	QNX	Chorus	VxWorks	pSOS	OS-9
Event Driven Dynamic Scheduling	X					
Scheduling Failure Detection and Handling	X					
Integrated Priority Scheduled Communications	X					
Location Transparent Distributed Services	X	X				
Distributed Inter Task Communication	X		X		X	
Publisher-Subscriber Model of Communication	X					
Dynamic Online Code Modification	X			X		
Dynamic Online Functional Reconfiguration	X					

When evaluating the performance of a distributed real time system, one typically considers the following areas: determinacy, worst case interrupt latency, context switch time, network latency, and service primitive times. Unfortunately, these terms do not have

unique meanings and standard measurement methodologies. The following sub-sections provide term definitions, measurement methodology and performance data.

9.3.1 Determinacy

A system engineer must be able to predict the worst-case timing behavior of a control application. In this context, it is important that a real-time system perform consistently regardless of the number of tasks, semaphores, or other resources allocated. Unfortunately, the performance of most operating systems is very sensitive to number of entities in the system. They use the term deterministic to mean that the execution times of their services can be calculated or predicted under a specific circumstance. However, this usage is in sharp contrast to the notion of deterministic meaning fixed cost under all situations and work loads.

An important design goal of DCOS was that all internal algorithms of real time services be fixed cost. Almost all DCOS real time primitives execute in a fixed amount of time regardless of the number of objects present in the system. The primary exception occurs when a task blocks while acquiring a resource and specifies a non-zero timeout interval. Other exceptions include obtaining a variable length memory block, remote object manipulation and some configuration services. Though these services have variable cost, they do not affect execution of other higher priority tasks, since they are preemptible.

In addition, the time required to service a clock tick interrupt is based upon the number of timeouts which expire at that tick. However, with a fine grained clock tick the average case expiry of timeouts is bounded to one. It should also be noted that even though real time primitives have fixed cost, being a event driven system the number of external events that can interrupt execution is unpredictable. But this is usually not a significant factor since the frequency of interrupts and interrupt service times are limited compared to the execution speed of processor.

9.3.2 Interrupt Latency

Interrupt latency is defined as the delay between receipt of an interrupt request and execution of the first user specified instruction in an interrupt service routine. Interrupts are critical component of event driven systems and it is necessary that they be acted upon as quickly as possible. The worst case interrupt latency for an real time operating system is based upon the following components:

1. the longest period of time interrupts are disabled by operating system
2. for some microprocessors, the length of longest instruction
3. the time required for processor to vector interrupt
4. the operating system overhead at the beginning of every interrupt service

The first component is irrelevant if an interrupt occurs when interrupts are enabled, although it must be included in a worst case analysis. The second and third components are specific to processor hardware and are not dependent on operating system. The first and second components are mutually exclusive and the longest of these two should be considered. The fourth component includes the time necessary for operating system to save registers and vector to user defined handler. Many real time operating systems report only the first component as their interrupt latency and ignore other components.

The definition used in this dissertation uses all four components to accurately reflect longest delay between receipt of an interrupt request and execution of first user specified instruction in an interrupt service routine. It should be noted that this definition does not include the components involved during simultaneous pending of multiple interrupts. For instance, a higher priority interrupt might have masked lower priority interrupts. These components are not accounted for since such information is available only with system engineer and the occurrence of simultaneous interrupts is dynamic.

9.3.3 Context Switch Time

Context switch is defined as the act of taking the processor from currently executing task and giving it to another task. This process involves selecting highest priority task that is ready, saving the hardware state of current task and restoring the

hardware state of new task. The hardware state of a task includes general purpose data registers, address registers, segment registers, control registers and paging registers. It should be noted that if either or both task(s) use floating point registers they need to be saved and/or restored.

A context switch is usually performed as part of a primitive's action or because of an interrupt. For example, if a task is unable to acquire a semaphore and blocks, a context switch is required to transfer control from the blocking task to a new task. Similarly an interrupt might make a higher priority task ready causing the current task to be preempted. In this case, the scheduling time for both preempting and preempted task should also be taken into account. Many real time operating systems report only saving and restoring of hardware state as context switch time. The remaining components should also be considered for accuracy.

9.3.4 Network Latency

Network latency is composed of following components: operating system overhead at sending end to map messages into communication packets, medium access latency, transmission latency and operating system overhead at receiving end to map packets into messages. Medium access latency is a variable component depending on network load and message priority. Transmission latency is a variable quantity depending on message length. Operating system overhead at receiving end is also variable quantity since a message may not be delivered at the receiving end immediately depending on its priority. Despite such variability network latency can be bounded and can be used to calculate round trip time and end to end communication delays.

9.3.5 Service Primitive Times

Service primitives are the application's interface to the operating system, and as such their execution times are critical in determining the performance of an application. For example, an application using a semaphore to protect a critical data structure should be aware of the time required to acquire and release a semaphore. In addition, a system

engineer can utilize service execution times to evaluate the performance of alternative means for synchronization and communication.

9.3.6 Methodology

All the times reported in following section except for the maximum period interrupts are disabled, were measured on the controller node 2 running on the 150MHz Pentium processor with 8KB integrated cache. This platform had a 2 wait state dynamic system memory and 256KB external cache. The integrated 64 bit clock counter of the Pentium processor was used to measure elapsed time with 6.67 nanosecond resolution. All sources of external and internal interrupts were enabled to reflect measurement under typical operation. The clock tick timer interrupt was configured for a period of 100μsec.

The times were measured for multiple invocations under various loads and situations, and averaged for better accuracy. The service primitive times were measured end to end including argument passing, raising and lowering of privilege level, stack switching and servicing by system agents. Times are provided for all primitives regardless of whether or not they are typically used in time critical code. For example, execution times are provided for configuration services such as object create and delete primitives, even though these are typically part of application initialization.

As noted earlier all real time services execute with fixed cost and hence the reported times of such services can be considered worst case performance. On the other hand some configuration services execute with variable cost and the reported times for such services should be considered average case performance. The maximum period interrupts are disabled was measured manually by summing the number of clock cycles required by each assembly language instruction within every block where interrupts were disabled. Zero wait state memory was assumed and the worst case times included instructions to disable and enable interrupts. The resulting clock cycles were converted to reflect times on a processor executing at 150MHz.

9.3.7 Performance Data

Tables 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.8, 9.9, 9.10 and 9.11 list the performance data for system agent, scheduler agent, timer agent, task agent, buffer pool memory agent, segmented heap memory agent, distributed shared memory agent, message port agent, semaphore agent and miscellaneous primitives, respectively. The significance of performance data reported in these tables is that these data can be used in designing distributed control applications. The data can be utilized to analyze and calculate timing information for operating system services used by a control application component. It can also be used to ensure that the timing constraints of a distributed application would be met. It should be noted that the performance data is implementation hardware specific and can only be used to analyze applications designed to run on the infrastructure described in previous sub-section. For the same reason, these data cannot be used for comparison with real time operating systems, unless the performance data for those systems were also measured on same hardware.

Table 9.2 - System Agent Primitives

Service Primitive	Time (μsec)	Remarks
SystemAgent.MapDualPortMem	11	Variable
SystemAgent.UnmapDualPortMem	19	Variable
SystemAgent.AddObjectIds	7	Variable
SystemAgent.AddObjects	6	Variable
SystemAgent.XferOwnership	6	
SystemAgent.Create	3	
SystemAgent.Delete	5	
SystemAgent.Probe	6	Overhead Only
SystemAgent.Open	5	Overhead Only
SystemAgent.Close	5	Overhead Only
SystemAgent.Control	7	Overhead Only
SystemAgent.Read	7	Overhead Only
SystemAgent.Write	9	Overhead Only
SystemAgent.InstallISR	3	
SystemAgent.SetInterruptLevel	3	
SystemAgent.SetInterruptLevel	3	Legal Limit

Table 9.3 - Scheduler Agent Primitives

Service Primitive	Time (μsec)	Remarks
SchedulerAgent.Yield	6	Overhead Only
SchedulerAgent.SetDeadline	12	Set Now Option
SchedulerAgent.ClearDeadline	5	
SchedulerAgent.GetTimingInfo	7	
SchedulerAgent.StartProfiling	3	
SchedulerAgent.StopProfiling	4	

Table 9.4 - Timer Agent Primitives

Service Primitive	Time (μsec)	Remarks
TimerAgent.GetSystemClock	11	
TimerAgent.SetSystemClock	8	
TimerAgent.NameToObject	5	
TimerAgent.AddObjects	8	Variable
TimerAgent.XferOwnership	5	
TimerAgent.Create	6	
TimerAgent.Delete	7	
TimerAgent.SetWatchDogAlarm	7	
TimerAgent.SetDayClockAlarm	10	
TimerAgent.SetPeriodicAlarm	7	
TimerAgent.CancelAlarm	3	

Table 9.5 - Task Agent Primitives

Service Primitive	Time (μsec)	Remarks
TaskAgent.NameToObject	5	
TaskAgent.AddProxyTasks	7	Variable
TaskAgent.AddObjects	17	Variable
TaskAgent.Create	200	Variable
TaskAgent.Delete	180	Variable
TaskAgent.ChangePriority	17	
TaskAgent.SleepFor	15	Overhead Only
TaskAgent.SleepTill	17	Overhead Only
TaskAgent.SendSignal	7	

Service Primitive	Time (μsec)	Remarks
TaskAgent.SetSignalHandler	3	
TaskAgent.SetSignalMask	2	
TaskAgent.Suspend	7	Overhead Only
TaskAgent.PostEvent	7	
TaskAgent.PendOnEvents	4	Available/Polling
TaskAgent.ThrowException	7	
TaskAgent.SetExceptionHandler	3	
TaskAgent.IdOfSelf	3	
TaskAgent.KillTask	7	Sending Kill Signal

Table 9.6 - Buffer Pool Memory Agent Primitives

Service Primitive	Time (μsec)	Remarks
BPMemAgent.NameToObject	5	
BPMemAgent.AddObjects	8	Variable
BPMemAgent.XferOwnership	5	
BPMemAgent.Create	10	Variable
BPMemAgent.Delete	10	
BPMemAgent.Allocate	5	
BPMemAgent.Return	5	
BPMemAgent.GetUsageInfo	6	

Table 9.7 - Segmented Heap Memory Agent Primitives

Service Primitive	Time (μsec)	Remarks
SHMemAgent.NameToObject	5	
SHMemAgent.AddObjects	12	Variable
SHMemAgent.XferOwnership	5	
SHMemAgent.Create	10	Variable
SHMemAgent.Delete	10	
SHMemAgent.Allocate	7	Variable
SHMemAgent.Return	6	
SHMemAgent.GetUsageInfo	6	

Table 9.8 - Distributed Shared Memory Agent Primitives

Service Primitive	Time (μsec)	Remarks
DSMemAgent.NameToObject	5	
DSMemAgent.AddSubscribers	29	Variable
DSMemAgent.AddObjects	14	Variable
DSMemAgent.XferOwnership	5	
DSMemAgent.Create	35	
DSMemAgent.Delete	40	
DSMemAgent.Open	10	
DSMemAgent.Close	5	
DSMemAgent.Read	6	Available/Polling
DSMemAgent.Write	22	

Table 9.9 - Message Port Agent Primitives

Service Primitive	Time (μsec)	Remarks
PortAgent.NameToObject	5	
PortAgent.AddObjects	11	Variable
PortAgent.XferOwnership	5	
PortAgent.Create	34	
PortAgent.Delete	27	
PortAgent.Send	13	
PortAgent.Receive	7	Available/Polling
PortAgent.AddMsgBufs	6	Variable

Table 9.10 - Semaphore Agent Primitives

Service Primitive	Time (μsec)	Remarks
SemaphoreAgent.NameToObject	5	
SemaphoreAgent.AddObjects	10	Variable
SemaphoreAgent.XferOwnership	5	
SemaphoreAgent.Create	31	
SemaphoreAgent.Delete	34	
SemaphoreAgent.Acquire	8	Available/Polling
SemaphoreAgent.Release	7	

Table 9.11 - Miscellaneous Timing Data

Service Primitive	Time (μsec)	Remarks
SEIAgent.LogMessage	14	
SEIAgent.LogData	20	
Interrupt Latency	1.3	
Context Switch Time	0.6	
Floating Point Context	1.4	Save and Restore
Static Scheduling	0.1	
Dynamic Scheduling	2	
Network Latency	80	

9.4 Functionality Tests

The functionality of the metamorphic control system was tested for proper implementation and operation using a number of distributed control applications. The process input/output requirements of these applications were satisfied through a multi-function I/O board developed specifically for this purpose. As shown in Fig. 9.3, this board has 8 analog to digital conversion channels with 12 bit resolution, 8 digital to analog conversion channels with 12 bit resolution, 16 digital inputs and 16 digital outputs. Every analog to digital conversion channel was equipped with a programmable amplifier providing gains of 1, 10, 100 and 1000. The completion of conversion triggers the configured processor interrupt, if enabled or can be polled by the processor, alternatively.

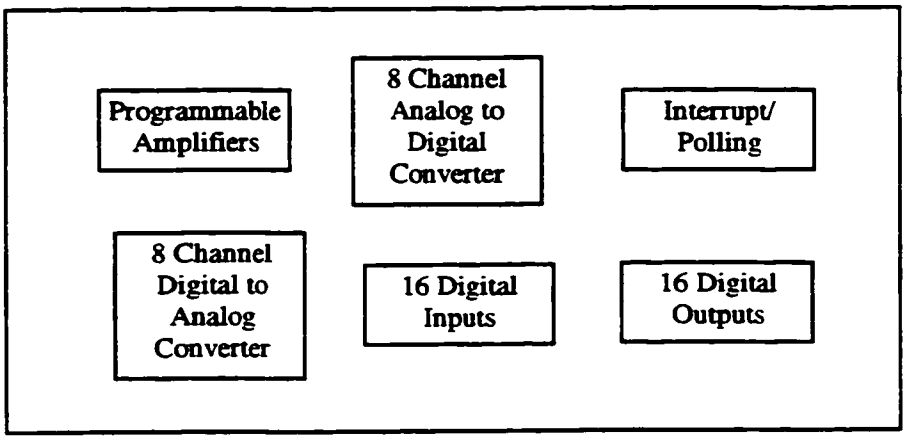


Figure 9.3: Multi-Function I/O Board

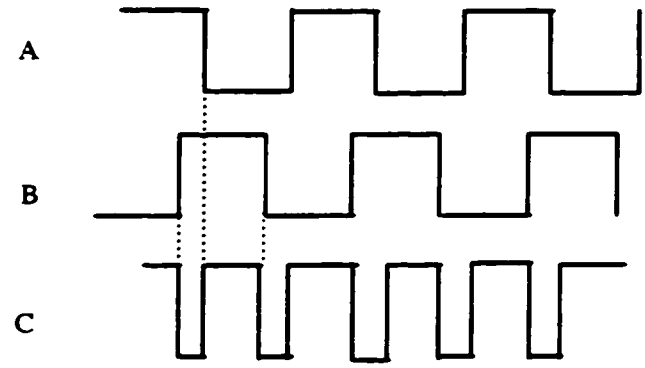
The various analog and digital I/O can be accessed through their respective 16 bit offset addresses from the base address of board. The board is designed for ISA system bus and can be plugged into any slot of this bus. The board can be mapped into the proper I/O address space of processor through configuration switches. Once configured, all inputs and outputs are accomplished through reading and writing at appropriate I/O addresses. A software device driver for this board was written and configured with the DCOS system agent for transparent access of functionality. The input/output function blocks such as ADC and DAC function blocks described in previous chapter, were accomplished by creating DCOS device object and manipulating it with primitives of system agent.

The following sub-sections describe three of the test cases that were used in functionality evaluation.

9.4.1 Test Case 1

The test case 1 was a simple frequency multiplication application. As shown in Fig. 9.4, two square wave signals out of phase with each other, were combined together through Boolean XOR operation to produce a frequency that is double that of source square waves. As may be noted, the duty cycle of the resulting wave is controlled by the phase shift between two source waves and the shape of the source waves, resulting in effective frequency multiplied software pulse width modulation.

Fig. 9.5 shows the function block components of control application used to generate the source and frequency multiplied waves. The IEC 1499 standard E_CYCLE and E_DELAY function blocks are used to generate periodic events, while the standard set dominant bistable function block E_SR and reset dominant bistable function block E_RS are used to generate source square waves. The initialization parameters for E_CYCLE and E_DELAY function blocks are used for setting up wave shapes and phase shift among them. As the name implies, the XOR function block performs Boolean XOR operation in an event driven fashion. The outputs of all waves are sent to digital output points on the multi-function I/O board through the DO function block.



C = A XOR B

Figure 9.4: Test Case 1 - Frequency Multiplication

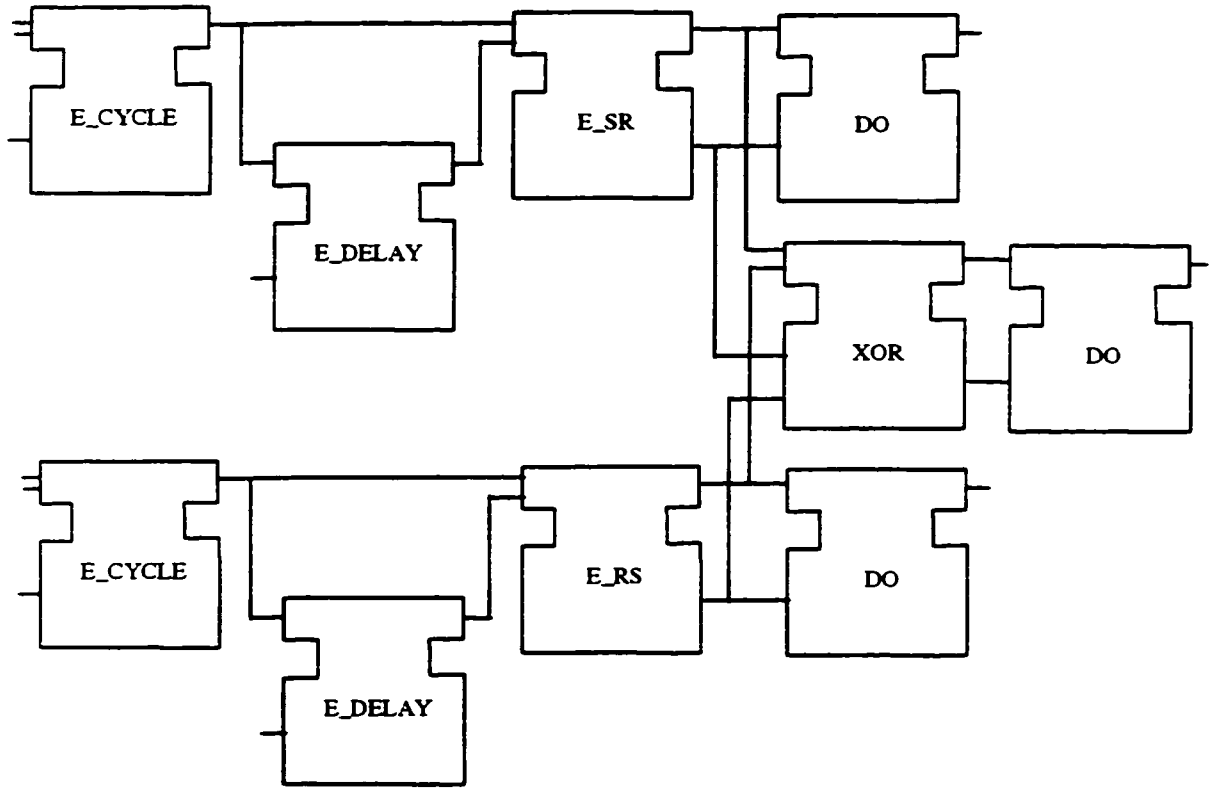


Figure 9.5: Test Case 1 - Frequency Multiplier

All the function block components of the frequency multiplier application were configured on node 2 of metamorphic control system for evaluation. The tests were carried out successfully by changing the shape and phase shift of source waves. The tests

were carried out at various frequencies of source waves ranging from 1Hz to 10000 Hz. The significance of these tests was to evaluate high speed event driven multi-tasking, timing and local messaging capabilities of the DCOS. The tests demonstrated these capabilities adequately.

9.4.2 Test Case 2

As shown in Fig. 9.6, this test case involved conventional regulatory PID control application, but implemented using event driven function blocks. As may be noted, the application is comprised of three components. The details of function blocks in this application were discussed in chapter 8.

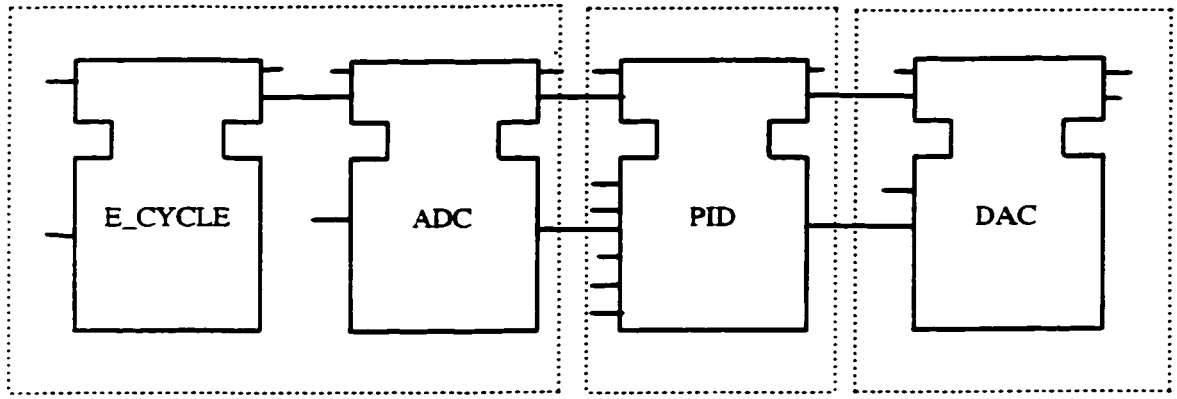


Figure 9.6: Test Case 2 - PID Application

Fig. 9.7 shows the various configurations used to test this application. In the instance 1 configuration, all 3 components of application were configured on node 2 with an application cycle period of 1 millisecond. In the instance 2 configuration, component 1 was configured on node 2, while the other two were configured on node 3. The application cycle period for this configuration was 2 milliseconds. In the instance 3 configuration, components 1, 2 and 3 were configured on nodes 1, 2 and 3, respectively. The application cycle period for this configuration was 5 milliseconds.

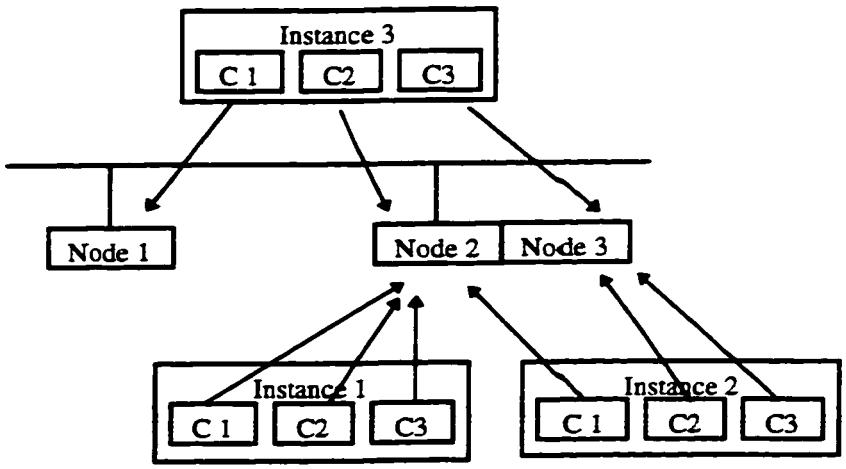


Figure 9.7: Test Case 2 - Distributed Configuration

The tests were carried out successfully, once each for every configuration. The significance of these tests was to evaluate location transparency, dynamic reconfiguration, client-server model of communication and trans-node consistent operation capabilities of the DCOS. These tests demonstrated the location transparency and dynamic reconfiguration since a single copy of application was used in multiple configurations. The client server model of communication was used to accomplish distributed control involving trans-node consistent operations.

9.4.3 Test Case 3

As shown in Fig. 9.8 and 9.9, this test case involved a distributed control application using the publisher-subscriber model of communication. The details of function blocks in this application were discussed in chapter 8. The objective of this test case is to demonstrate fault tolerant control using active backup redundancy techniques and voting mechanisms. In this case, it is accomplished through two copies each of publisher and subscriber components. Every subscriber component receives input data from both publisher components (voting). The subscriber component decides which one to use through fuzzy reasoning and computes the control output through fuzzy logic. The significance of two copies of components is that at least one copy will function properly in case one of them fails (active backup redundancy).

Figure 9.10 shows the configurations used in evaluation. In the first instance, one copy each of publisher and subscriber were instantiated on nodes 1 and 2, and active backup redundant copies were instantiated on nodes 2 and 3, respectively. The application cycle period for this configuration was 6 milliseconds. In the second instance, one copy each of publisher and subscriber were instantiated on nodes 1 and 3, and active backup redundant copies were instantiated on nodes 2 and 1, respectively. The application cycle period for this configuration was 8 milliseconds.

The tests were carried out successfully once each for every configuration. The significance of these tests was to evaluate publisher-subscriber model communication, location transparency, dynamic reconfiguration and trans-node operational capabilities of DCOS. The tests demonstrated trans-node distributed operations primarily through publisher-subscriber model of communication. However as mentioned earlier, the most important aspect of this test case was to evaluate fault tolerant control through active backup redundancy. To accomplish this one of the components was failed at random in each configuration. The redundant component ensured proper functioning, thereby demonstrating fault tolerance. It should be noted that N-way redundant fault tolerant control can also be implemented through similar techniques.

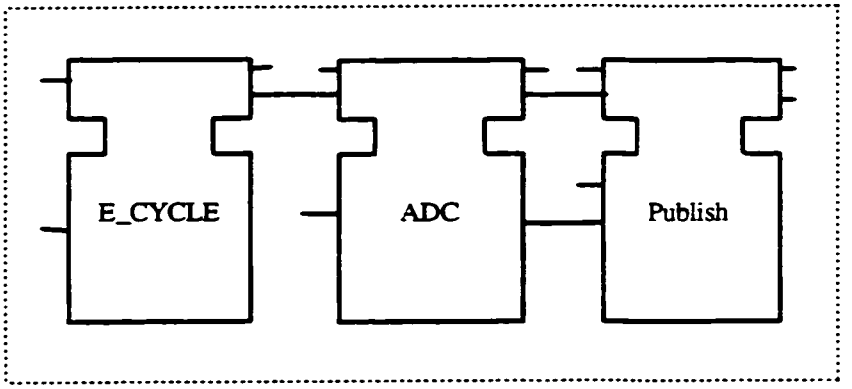


Figure 9.8: Test Case 3 - Publisher Component

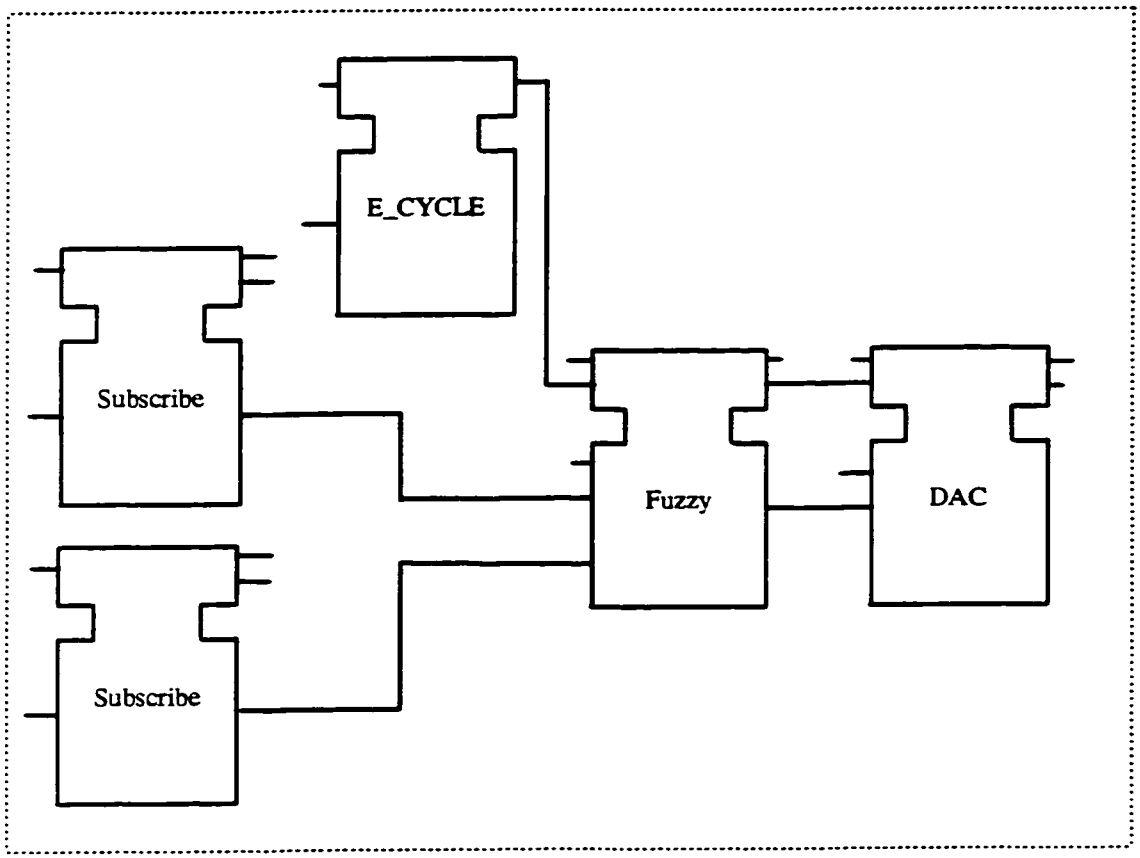


Figure 9.9: Test Case 3 - Subscriber Component

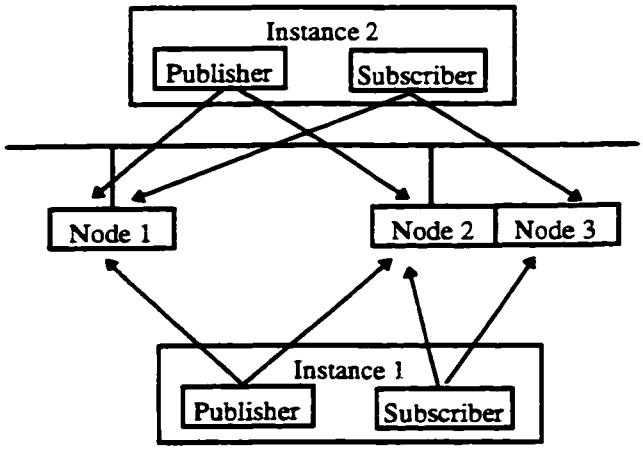


Figure 9.10: Test Case 3 - Distributed Configuration

9.5 Summary

In this chapter, the implementation and evaluation of a prototype metamorphic control system was presented. The implementation details included description of controller platforms and networking infrastructure. This was followed by discussions on measurement of performance, methods of gathering timing data and other time critical aspects that affect applications design such as determinacy, interrupt latency and context switch times. The timing data for various operating system service primitives were also provided and the significance of these were discussed. Finally, some of the test cases used in the functionality evaluation of the implemented metamorphic control system were presented.

Chapter 10

Contributions and Recommended Future Work

10.1 Summary

Real time control of autonomous and cooperative holonic systems requires a radically different approach from traditional unit level control systems. Holonic systems combine knowledge and material processing, and are evolutionary in nature to accommodate changing requirements. Evolutionary systems require both static and dynamic reconfigurability of their control systems. The control requirements of holonic resources move away from being that of centralized to one of being distributed, necessitating a system level approach. The dynamics of distributed control result in complex behavior and can only be handled through an event driven control system. Additionally, autonomous systems require incorporation of intelligence into their control systems.

This dissertation addressed the engineering of software centric open architecture control systems for cooperating networks of distributed autonomous resources. A comprehensive software agent based metamorphic control architecture was developed for dynamically reconfigurable distributed multi-sensor based holonic systems. This reference architecture defined vital system wide components at hardware and software levels. It uses IEC 1499 function block specification standard for modeling distributed applications. The critical issues associated with development of dynamically reconfigurable event driven control systems were also identified.

A prototype metamorphic control system developed as a proof of concept implementation was presented. In this system, the core mechanisms required for metamorphic control of distributed systems were incorporated into a distributed real time operating system. These mechanisms provide flexible and extensive functionality for implementing event driven control systems. One feasible method of developing distributed

application software from the function block specification was described. A system engineering interface developed to address remote software development, configuration and maintenance requirements of distributed control system was presented.

The performance of the implemented system was evaluated and extensive timing data pertinent to design of distributed applications were presented. This prototype system has adequately demonstrated the feasibility and usefulness of metamorphic control architecture for distributed multi sensor based holonic systems. It should be noted that this metamorphic control system is not restricted to "green field" applications, but can be interfaced with "legacy systems" as well.

10.2 Research Contributions

The issues associated with design and development of intelligent control systems for distributed, autonomous and cooperative manufacturing resources such as holonic systems have not been addressed by earlier research. The objective of the research presented in this dissertation was to develop a software centric open architecture metamorphic control system for dynamically reconfigurable distributed multi-sensor based real time systems. This has been achieved and this research is expected to have following ramifications in the field of industrial controller research and development.

1. Earlier research in related fields of interest such as distributed computing, real time operating systems, formal specification techniques and intelligent autonomous control have yielded several significant results, albeit originally meant for different applications. The integration of these results using a system level approach with focus on dynamically reconfigurable distributed control systems has remained unaddressed. By defining an unified, agent based metamorphic control architecture this research has made an original contribution in the distributed control area. Further, the critical issues involved with implementation of metamorphic control systems have also been identified.

2. By developing a new distributed real time operating system incorporating the core functionality required for metamorphic control, this research makes the following contributions to real time operating system research:
    - It has an unique distributed object based architecture and deterministic implementation that has implications for operating system design.
    - A novel device driver management technique facilitating online extension of functionality has been developed.
    - A new dynamic mixed priority scheduling algorithm has been developed.
    - A new integrated priority communication scheduling mechanism has been developed.
3. This research is the only known work at the time of writing this dissertation, to address significant distributed control issues associated with the implementation of the emerging IEC 1499 function block specification standard. The incorporation of the requisite functionality into the developed operating system led to the identification of several important design and implementation issues that were communicated to the IEC 1499 Standards Committee for consideration.

10.3 Future Work

The proof of concept metamorphic control system presented in this dissertation breaks new ground but still leaves many features to be desired. The development of the following tools should be particularly addressed in future work:

- *A generic visual programming interface for development of application software from the IEC 1499 function block specification.* This environment should be able to generate code automatically with equivalent functionality of function blocks from graphical specification. Such an environment would reduce much of the tedium associated with textual code development. The issues associated with such an interface includes development of a programming methodology that will produce consistent code for a wide variety of situations and maintain data consistency under asynchronous event driven execution and communication.

- *A sophisticated system engineering interface with all remote management capabilities required for class-2 user reprogrammable implementation as specified in the IEC 1499 standard.* This interface should support remote debugging and performance monitoring tools for function block code development and maintenance. It should also support traditional human machine interface elements for development of customized graphical operator interfaces.
- *A system analysis tool to verify if a distributed application under particular configuration would satisfy critical timing constraints.* Such a tool must use real time performance data and current load on system hardware to analyze and/or simulate new configuration. It should also support expert knowledge and intelligent reasoning mechanisms to suggest improvements and modifications in computation allocations and possible courses of actions.
- *A library of mechanisms for development of application specific intelligent execution control agents to provide higher order autonomy and cooperation.* Such mechanisms should include hybrid reactive and deliberative reasoning techniques to incorporate soft real time goal directed behavior to execution control agents. In order to be compatible with hard real time distributed control agents, the execution control agents will also have to use the IEC 1499 function block standard for behavior specification.
- *A prototype manufacturing application involving autonomous and cooperative holonic resources.* Such an application would act as a test bed for two purposes. First, it can be used to experiment, refine and demonstrate capabilities of metamorphic control system. Second, it can be used as a benchmark for comparison with other types of traditional control paradigms.

References

[Albus91] J. S. Albus, "Outline for a Theory of Intelligence," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 21, No. 3, 1991, pp. 473-509.

[Alur89] R. Alur and T. A. Henzigner, "A Really Temporal Logic," *In Proceedings of 30th Annual Symposium on Foundations of Computer Science*. 1989, pp. 164-169.

[Arvin89] K. Arvind, "A New Probablistic Algorithm for Clock Synchronization," *In Proceedings of Real Time Systems Symposium*. 1989, pp. 330-339.

[Baet91] J. C. M. Baeten and J. A. Bergstra, "Real Time Process Algebra," *Formal Aspects of Computing*, Vol. 3, No.2, 1991, pp. 142-188.

[Bala96] S. Balasubramanian and D. H. Norrie, "Intelligent Manufacturing System Control," *In Proceedings of 1996 Canadian Conference on Electrical and Computer Engineering*, 1996, pp. 570-573.

[Bala97] S. Balasubramanian, *DCOS Programming Manual*, Division of Manufacturing Engineering, The University of Calgary, 1997.

[Brooks86] R. A. Brooks, "A Robust Layered Control System for a Mobile Robot," *IEEE Journal of Robotics and Automation*, Vol. 2, No. 1, 1986, pp. 14-23.

[Brooks91a] R. A. Brooks, "Intelligence without Reason," *In Proceedings of 12th International Joint Conference on Artificial Intelligence*, Menlo Park, Morgan Kaufmann, 1991, pp. 569-595.

[Brooks91b] R. A. Brooks, "Intelligence without Representation," *Artificial Intelligence*, Vol. 47, 1991, pp. 139-159.

[Brooks91c] R. A. Brooks, "Elephants Don't Play Chess." In P. Maes, Editor, *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, London, The MIT Press, 1991, pp. 3-15.

[Bryan88] E. A. Bryan, and L. A. Bryan, *Programmable Controllers - Theory and Implementation*, Industrial Text Co., 1988.

[Chat92] Andy Chatha and Chantal Polsonetti, "Fieldbus Standard: We Need a Winner," *Instrumentation & Control Systems*, October, 1992, pp. 29-31.

[Chen90a] M. Chen and K. Lin, "Dynamic Priority Ceilings: A Concurrency Control Protocol for Real Time Systems," *Journal of Real Time Systems*, Vol. 2, 1990, pp. 325-346.

[Chen90b] Y. Chen and T. Chen, "Implementing Fault Tolerance via Modular Redundancy with Comparison," *IEEE Transactions on Reliability*, Vol. 39, No. 2, 1990, pp. 150-173.

[Cheng88] Sheng Chang Cheng, John A. Stankovic, and Krithi Ramamritham, "Scheduling Algorithms for Hard Real Time Systems - A Brief Survey," *IEEE Computer*, 1988, pp. 150-173.

[Chorus96] -, *STREAM API-v2 Kernel Architecture and API Specification*, Version 1.0, Chorus Systems Inc., 1996.

[Chris94a] James H. Christensen, Douglas Norrie and Christoph Schaeffer, "Material Handling Requirements in Holonic Manufacturing Systems," *In Proceedings of 2nd International Conference on Material Handling Research*, Michigan, 1994, pp.1-22.

[Chris94b] James H. Christensen, "Holonic Manufacturing Systems: Initial Architecture and Standards Directions," *In Proceedings of First European Conference on Holonic Manufacturing Systems*, Hannover, Germany, 1994, pp. 1-20.

[Connell92] J. H. Connell, "SSS: A Hybrid Architecture Applied to Robot Navigation," *In Proceedings of International Conference on Robotics and Automation*, 1992, pp. 2719-2724.

[Crist89] F. Cristian, "Probablistic Clock Synchronization," *Distributed Computing*, Vol. 3, 1989, pp. 146-158.

[Deme95] L. Demeestere, H. Thielemans, and H. Van Brussel, "HEDRA: Heterogeneous Distributed Real-Time Architecture," *In Proceedings of 3rd*

Workshop on Algorithms and Architectures for Real-Time Control, Ostend, Belgium, 1995, pp. 517-524.

[Dert89] M. L. Dertouzos and A. K. Mok, "Multiprocessor Online Scheduling of Hard Real Time Tasks," *IEEE Transactions on Software Engineering*, Vol. 15, No. 12, 1989, pp. 1497-1506.

[Dij68] Edsger W. Dijkstra, "The Structure of THE-Multiprogramming System," *Communications of the ACM*, Vol. 11, No. 5, May 1968, pp. 345-346.

[Doty95] K. L. Doty and A. B. Ghannam, "Controlling Situated Agent Behaviors with Consistent World Modelling and Reasoning," *In Proceedings of AAAI Spring Symposium on Lessons Learned from Implemented Software Architectures for Physical Agents*, 1995, pp. 50-56.

[Earl92] Earl Chafin, "Multiloop Controllers Become Multifunctional," *Instrumentation & Control Systems*, August, 1992, pp. 51-54.

[ELF94] Intel, *Executable and Linking Format*, Portable Format Specification V1.1, 1994.

[Eric96] Eric Jackson and David Eddy, *Design and Implementation Methodology for Autonomous Robot Control Systems*, Technical Paper, International Submarine Engineering, Ltd., Vancouver, 1996.

[Erik96] C. Eriksson, J. Maki-Turia, L. Post, M. Gustafsson, J. Gustafsson, K. Sandstrom, and E. Brorsson, "An Overview of RealTimeTalk, a Design Framework for Real Time Systems," *Journal of Parallel and Distributed Computing*, Vol. 36, No.1, 1996, pp. 66-80.

[Ezhil86] P. Ezhilselvan and S. K. Shrivatsava, "A Characterization of Faults in Systems," *In Proceeedings of 5th Symposium on Reliability in Distributed Software and Database Systems*, 1986, pp. 215-222.

[Firby94] R. J. Firby, "Architecture, Representation and Integration: An Example From Robot Navigation," *In Proceedings of AAAI Fall Symposium on Control of the Physical World by Intelligent Agents*, 1994, pp. 55-59.

[Gat91] E. Gat, "Integrating Planning and Reacting in a Heterogeneous Asynchronous Architecture for Mobile Robots," *SIGART Bulletin*, No. 2, 1991, pp. 70-74.

[Gaud96] D. Gaudreau and P. Freedman, "Temporal Analysis and Object-oriented Real-Time Software Development: A Case Study with ROOM/ObjecTime," *In Proceedings of 1996 IEEE Real-Time Technology and Applications Symposium*, 1996, pp. 110-118.

[Geha91] N. Gehani and K. Ramamrithm, "Real Time Concurrent C: A Language for Programming Real Time Systems," *Journal of Real Time Systems*, Vol. 3, No. 4, 1991, pp. 377-405.

[Gertz93] M. Gertz, D. Stewart, and P. Khosla, "A Software Architecture Based Human Machine Interface for Reconfigurable Sensor Based Control Systems," *In Proceedings of IEEE International Symposium on Intelligent Control*, 1993, pp. 75-80.

[Ghez91] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezze, "A Unified High Level Petri Net Formatism for Time Critical Systems," *IEEE Transactions on Software Engineering*, Vol. 17, No. 2, 1991, pp. 12-17.

[Gon91] G.H. Gonnet and R. Baeza-Yates, *Handbook of algorithms and data structures*, Addison-Wesley, 1991.

[Gud90] O. Gudmundsson, D. Mosse, A. Agrawala, and S. Tripathi, "Maruthi: A Hard Real Time Operating System," *In Proceedings of 2nd Workshop on Experimental Distributed Systems*, 1990, pp. 29-34.

[Hare96] D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts," *ACM Transactions on Software Engineering and Methodology*, Vol. 5, No. 4, 1996, pp. 293-333.

[Harr96] R. Harrison, C.D. Wright, A. H. Booth, and A. J. Carrott, "IMDC: An integrated environment for the design and control of manufacturing machines," *IEE Colloquium Digest*, Issue 42, 1996, pp. 4/1-4/8.

[Hewitt77] C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages," *Artificial Intelligence*, Vol. 8, No. 3, 1977, pp. 323-364.

[Huang96] H. M. Huang, "An Architecture and a Methodology for Intelligent Control," *IEEE Expert*, April, 1996, pp. 46-55.

[IEC93] IEC 1131-3, *Programmable Controllers - Programming Languages*, International Electrotechnical Commission, 1993.

[IEC97] IEC 1499-1, *Function Blocks for Industrial Process Measurement and Control Systems - Architecture*, Committee Draft 10, International Electrotechnical Commission, 1997.

[IEEE83] IEEE 729, *IEEE Standard Glossary of Software Engineering Terminology*, Institute of Electrical and Electronic Engineers, 1983.

[IEEE93] IEEE 1003.1b, *IEEE Standard for Information Technology - Portable Operating System Interfaces - System Application Programming*, Institute of Electrical and Electronic Engineers, 1993.

[IP81] RFC-791, *Internet Protocol*, Internet Engineering Task Force, 1981.

[Ishi92] Y. Ishikawa, H. Tokuda, and C. Mercer, "An Object Oriented Real Time Programming Language," *IEEE Computer*, Vol. 25, No. 10, 1992, pp. 66-73.

[ISI93] -, *pSOS Operating System - User Manual*, Integrated Systems Inc., 1993.

[ISO84] ISO 7498, *Information Processing Systems - Open Systems Interconnection*, International Standards Organization, 1984.

[ISO90] ISO 9506, *Industrial Automation Systems - Manufacturing Message Specification*, International Standards Organization, 1990.

[Jah88] F. Jahanian and A. K. Mok, "Modechart: A Specification Language for Real Time System," *IEEE Transactions on Software Engineering*, Vol. 14, No. 1, 1988, pp. 10-15.

[Jen90] E. Jensen and J. Northcutt, "ALPHA: A Non-properietary OS for Large, Complex, Distributed Real Time Systems," *In Proceedings of 2nd IEEE*

Workshop on Experimental Distributed Systems, October, 1990, pp. 20-28.

[John95] D. Johnson, "Looking over the bus systems," *Control Engineering*, Vol. 42, No. 13, December, 1995, pp. 56-64.

[Kenn91] K. B. Kenny and K. J. Lin, "Building Flexible Real Time Systems Using the Flex Language," *IEEE Computer*, Vol. 24, No. 5, 1991, pp. 70-78.

[Koest71] A. Koestler, *The Ghost in the Machine*, 1971, ISBN 0-14-019192-5.

[Kop87] H. Kopetz and W. Ochsenreiter, " Clock Synchronization in Distributed Real Time Systems," *IEEE Transactions on Computers*, Vol. 36, No. 8, 1987, pp. 933-940.

[Kop89] H. Kopetz, A. Damm, C. Koza, M.Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed Fault Tolerant Real Time Systems: The MARS Approach," *IEEE Micro*, Vol. 9, NO. 1, 1989, pp. 25-40.

[Kop93] Hermann Kopetz, "Scheduling," In Sape Mullender, Editor, *Distributed Systems*, ACM Press and Addison-Wesley, 1993, pp. 491-509.

[Kop94] H. Kopetz and G. Grunsteidl, "TTP - A Protocol for Fault Tolerant Real Time Real Time Systems," *IEEE Computer*, Vol 27, No. 1, 1994, pp. 14-23.

[Kuro88] J. F. Kurose, M. Schwartz, and Y. Yemini, "Controlling Window Protocols for Time Constrained Communication in Multiple Access Networks," *IEEE Transactions on Communications*, Vol. 36, No. 1, 1988, pp. 41-49.

[Lala91] J. Lala, R. Harper, and L. Alger, "A Design Approach for Ultrareliable Real Time Systems," *Computer*, Vol. 24, No. 5, 1991, pp. 12-22.

[Lamp82] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals' Problem," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, 1982, pp. 382-401.

[Lamp85] L. Lamport and P. M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults," *Journal of the ACM*, Vol. 32, No. 1, 1985, pp52-78.

[Lapri88] J. C. Laprie, "Dependability: A Unifying Concept for Reliable Computing and Fault Tolerance," *Resilient Computing Systems*, Vol. 2, T. Anderson Editor, Collins and Wiley, 1988.

[Law83] E. L. Lawler, "Recent Results in the Theory of Machine Scheduling," In A. Bachem et. al., Editor, *Mathematical Programming: The State of the Art*, Springer Verlag, 1983, pp. 202-233.

[Leho87] J. P. Lehoczky, L. Sha, and J. K. Strosnider, "Enhanced Aperiodic Responsiveness in Hard Real Time Environments," *In Proceedings of 8th IEEE Real Time Systems Symposium*, December 1987, pp. 261-270.

[Lim91] C. C. Lim, L. j. Yao, and W. Zhao, "A Comparative Study of Three Token Ring Protocols for Real Time Communications," *In Proceedings of 11th IEEE International Conference on Distributed Computing Systems*, 1991, pp. 308-317.

[Liu73] C. W. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment," *Journal of the ACM*, Vol. 20, No. 1, 1973, pp. 46-61.

[Lo90] H. Y. Lo, L. P. Ju, and C. C. Su, "General Version of Reconfiguration N Modular Redundancy System," *In IEE Proceedings on Circuits, Devices and Systems*, 1990, pp. 137-145.

[Lund84] J. Lundelius-Welch and N. Lynch, "An Upper and Lower Bound for Clock Synchronization," *Information and Control*, Vol. 62, 1984, pp. 190-204.

[Lynch89] N. A. Lynch and M. R. Tuttle, "An Introduction to Input/Output Automata," *CWI Quarterly*, Vol. 2, No. 3, 1989, pp. 219-246.

[Maes91] P. Maes, *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, London, The MIT Press, 1991.

[Mataric92] M. J. Mataric, "Integration of Representation into Goal Driven Behavior Based Robots," *IEEE Transactions on Robotics and Automation*, Vol. 8, June 1992, pp. 304-312.

[MC95] - , "PLCs Suit Up for Robot Control," *Machine Design*, April, 1995, pp. 160-162.

[McMa95] R. McMahon, "Sensor bus technology: Revolutionizing factory process control," *Semiconductor International*, Vol. 18, No. 8, July, 1995, pp. 123-126.

[Micro91] -, *OS-9 Version 2.4 Operating System - User Manual, Microware Systems Corp.*, 1991.

[Mok83] A. K. Mok, *Fundamental Design Problems of Distributed Systems for the Hard Real Time Environment*, Ph.D. Thesis, M.I.T., 1983.

[Musl93] D. J. Musliner, E. H. Durfee, and K. G. Shin, "CIRCA: A Cooperative Intelligent Real Time Control Architecture," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 23, No. 6, 1993, pp. 1561-1574.

[Nil96] K. Nilsen, "Java for Real Time," *Real Time Systems*, Vol.11, No.2, 1996, pp. 197-205.

[Norrie94] Douglas H. Norrie, *A Vision of the Next Generation of Manufacturing System*, Internal Document, Division of Manufacturing Engineering, The University of Calgary, 1994.

[Ohare96] G. M. P. O'Hare and N. Jennings, *Foundations of Distributed Artificial Intelligence*, John Wiley, 1996.

[OMAC94] Chrysler, Ford and GM, *Requirements of Open, Modular Architecture Controllers for Applications in the Automotive Industry*, Specification Document V1.1, 1994.

[OSACA96] ESPRIT III Project 6379, *Open System Architecture for Controls within Automation Systems*, Final Report V1.4, 1996.

[OSEC95] OSEC Consortium, *Open System Environment Controller*, Specification Document V2.0, 1995.

[Owen95] J. Owen, "Open Controllers," *Manufacturing Engineering*, November 1995, pp. 53-60.

[Pack97] R. T. Pack, M. Wilkes, G. Biswas, and K. Kawamura, "Intelligent Machine Architecture for Object Based System Integration," *In Proceedings of 1997 IEEE/ASME International Conference on Advaced Intelligent Mechatronics*, 1997, pp. 11-16.

[Panz93] F. Panzeri and R. Davoli, "Real Time Systems: A Tutorial," In Lorenzo Donatiello and Randolph Nelson, Editors, *Performance Evaluation of Computer and Communication Systems*, Lecture Notes in Computer Science, Vol. 729, Springer Verlag, 1993, pp.435-462.

[Park95] J. Park, S. Birla, K. G. Shin, and Z. J. Park, "An Open Architecture Testbed for Real-time Monitoring and Control of Machining Systems," *In Proceedings of the American Control Conference*, Seattle, 1995, pp. 200-204.

[Plien92] P. Plienvaux, "An Improved Hard Real Time Scheduling for the IEEE 802.5," *Journal of Real Time Systems*, Vol. 4, No.2, 1992, pp. 99-112.

[Proc93] F. M. Proctor, and J. Michaloski, *Enhanced Machine Controller Architecture Overview*, NIST Technical Report 5331, December 1993.

[QNX93] -, *QNX 4 Operating System - System Architecture*, QNX Software Systems Ltd., 1993.

[Ram87] K. Ramamritham, "Channel Characteristics in Local Area Hard Real Time Systems," *Computer Networks and ISDN Systems*, Vol. 13, No. 1, 1987, pp. 3-13.

[Raj93] S. C. V. Raju, *An Automatic Verification Technique for Communicating Real Time State Machines*, Technical Report UW-CSE-93-04-08, Department of Computer Science and Engineering, University of Washington, 1993.

[Ren95] S. Ren and G. A. Agha, "RTsynchronizer: Language Support for Real Time Specifications in Distributed Systems," *SIGPLAN Notices*, Vol.30, No.11, pp. 50-59.

[Rose97] J. K. Rosenblatt, "DAMN: A Distributed Architecture for Mobile Navigation," *Journal of Experimental & Theoretical Artificial Intelligence*, Vol. 9, No. 2/3, 1997, pp. 339.

[RTE96] -, "Real Time Operating Systems," Real Time Engineering Magazine, Vol. 3, No. 2, 1996, pp. 24-27.

[Sak89] K. Sakamura and R. Sprague, "The TRON Project," *Byte*, April, 1989, pp. 292-301.

[Sarma95] S. Sarma, R. Narayanaswami, S. Schofield, and P. Wright, "Machine Tool Open System Advanced Controller for Precision Manufacturing (MOSAIC-PM)," *In Proceedings of NSF Design and Manufacturing Grantees Conference*, La Jolla, 1995, pp. 151-152.

[Schl83] R.D. Schlichting and F. Schneider, "Fail Stop Processors: An Approach to Designing Fault Tolerant Computer Systems," *ACM TOCS*, Vol. 1, No. 3 1983, pp. 222-238.

[Schn87] F. Schneider, *Understanding Protocols for Byzantine Clock Synchronization*, Technical Report 87-859, Cornell University, 1987.

[Schn95] S. Schneider, V. Chen, J. Steele, and G. Pardo-Castellote, "The ControlShell component-based real-time programming system, and its applications to the Marsokhod Martian Rover," *In Proceedings of ACM Workshop on Languages, Compilers, and Tools for Real Time System*, La Jolla, 1995, pp. 146-155.

[Sha90] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real Time Synchronization," *IEEE Transactions on Computers*, Vol. 39, No. 9, 1990, pp. 1175-1185.

[Shin90] K. G. Shin and C. J. Hou, "Analysis of Three Contention Protocols in Distributed Real Time Systems," *In Proceeedings of IEEE Real Time Systems Symposium*, 1990, pp. 136-145.

[Shin91] K. G. Shin, "HARTS: A Distributed Real Time Architecture," *Computer*, Vol. 24, No. 5, 1991, pp. 25-35.

[Siew84] D. P. Siewiorek, "Architecture of Fault Tolerant Computers," *Computer*, Vol. 17, August, 1984, pp. 9-17.

[SMP95] Intel, *MultiProcessor Specification*, Technical Document Version 1.4, 1995.

[Sprunt89] Brinkley Sprunt, Lui Sha, and John Lehoczky, "Aperiodic Task Scheduling for Hard Real Time Systems," Journal of Real Time Systems, Vol. 1, 1989, pp. 27-60.

[Srik87] T. K. Srikanth and S. Toueg, "Optimal Clock Synchronization," *Journal of the ACM*, Vol. 34, No. 1, pp. 626-645.

[Stan88] John A. Stankovic, "Misconceptions About Real Time Computing," *IEEE Computer*, Vol. 21, No. 10, October 1988, pp. 10-19.

[Stan91] J. A. Stankovic and K. Ramamrithm, "The Spring Kernel: A New Paradigm for Real Time Systems," *IEEE Software*, Vol. 8, No. 3, 1991, pp. 62-72.

[Stan93] John. A. Stankovic and Krithi Ramamritham, "Scheduling," *In Advances In Real-Time Systems*, IEEE Computer Society Press, 1993, pp. 47-52.

[Sten97] Jon Stenerson and Kelly Curran, *Computer Numerical Control : Operation and Programming*, Prentice Hall, 1997.

[Stew91] D. B. Stewart and P. K. Khosla, "Real Time Scheduling of Dynamically Reconfigurable Systems," *In Proceeedings of 1991 International Conference on Systems Engineering*, Dayton, August 1991, pp. 139-142.

[Stew94] D. B. Stewart and P. K. Khosla, "The Chimera Methodology: Designing Dynamically Reconfigurable Real-Time Software Using Port-Based Objects," *In Proceedings of 1st IEEE Workshop on Object Oriented Real Time Dependable Systems*, Dana Point, 1994, pp. 46-55.

[Stros88] J. K. Strosneider, T. Marchok, and J. Lehoczky, "Advanced Real Time Scheduling Using the IEEE 802.5 Token Ring," *In Proceedings of IEEE Real Time Systems Symposium*, 1988, pp.42-52.

[Stros89] J. K. Strosneider and T. Marchok, "Responsive, Deterministic IEEE 802.5 Token Ring Scheduling," *Journal of Real Time Systems*, Vol. 1, No. 2, 1989, pp. 133-158.

[Tak92] K. Takashio and M. Tokoro, "DROL: An Object Oriented Programming Language for Distributed Real Time Systems," *In Proceedings of OOPSLA'92 Conference*, 1992, pp. 276-294.

[Tan87] Andrew S. Tanenbaum, *Operating systems : design and implementation*. Prentice-Hall, 1987.

[Tok89] H. Tokuda and C. Mercer, "ARTS: Adistributed Real Time Kernel." *ACM Operating Systems Review*, Vol. 23, No. 3, 1989, pp. 29-53.

[Tok90] H. Tokuda, T. Nakajima, and P. Rao, "Real Time Mach: Towards Predictable Real Time Systems," *In Proceedings of 1990 USENIX Mach Workshop*, 1990, pp. 35-45.

[Val92] A. Valenzano, C. Demartini, and L. Ciminiera, *MAP and TOP Communications*, Addison Wesley, 1992.

[Varg87] G. Varghese and A. Lauck, "Hashed and Hierarchical Timing Wheels: Data Structures for the efficient Implementation of a Timer Facility," *In Proceedings of 11th ACM Symposium on Operating System Principles*, 1987, pp. 171-180.

[Wayn91] Wayne Labs, "DCS Technology Update," *Instrumentation & Control Systems*, October, 1991, pp. 27-31.

[WRS94] -, *VxWorks 5.2 Operating System - User Manual*, Wind River Systems, 1994.

[Yello96] L. Yellowley, R. Ardekani, L. Yang, and R.J. Seethaler, "Development of robust extensible architectures for machine-tool control," *In Proceedings of SPIE Meeting on Open Architecture Control Systems and Standards*, Boston, 1996, pp. 72-83.

[Zhao88] W. Zhao, J. A. Stankovic, and K. Ramamrithm, "A Multi Access Window Protocol for Time Constrained Communications," *In Proceeedings of 8th*

IEEE International Conference on Distributed Computing Systems, 1988, pp. 384-392.

[Znati91] T. Znati, "Deadline Driven Window Protocol for Transmission of Real Time Traffic," *In Proceedings of 10th IEEE International Conference on Computers and Communications*, 1991, pp. 667-673.