A COMMON INTERFACE REAL-TIME MULTIPROCESSOR OPERATING SYSTEM FOR EMBEDDED SYSTEMS

THESIS

Michael S. Rottman
Captain, USAF

AFIT/GCE/ENG/91M-04

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCE/ENG/91M-04

A COMMON INTERFACE REAL-TIME MULTIPROCESSOR OPERATING SYSTEM FOR EMBEDDED SYSTEMS

THESIS

Michael S. Rottman
Captain, USAF

AFIT/GCE/ENG/91M-04

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; Distribution Unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCE/ENG/91M-04	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code)	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Flight Dynamics Directorate Control Systems Dev Branch 8b. OFFICE SYMBOL (If applicable) WL/FIGL	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB, OH 45433-6553	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification) (Unclassified)
A Common Interface Real-Time Multiprocessor Operating System for Embedded Systems

12. PERSONAL AUTHOR(S)
Michael S. Rottman, Captain, US Air Force

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS Thesis	FROM _____ TO _____	910304	226

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Real-Time Software, Operating Systems, Parallel Processing, Software Development
09	02		
23	06		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Advisor: Dr Thomas Hartrum, PhD
Professor, Department of Electrical and Computer Engineering
(see reverse)

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr Thomas Hartrum	22b TELEPHONE (Include Area Code) (513) 255-3708 / 22c OFFICE SYMBOL AFIT/ENG

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

Large real-time applications such as aerospace avionics systems, battle management, and factory automation place many demands and constraints on the computing system not found in other applications. Software development is hindered by software dependence on the computer architecture and the lack of portability between systems. This thesis specifies and designs a real-time multiprocessor operating system (RTMOS) that implements a consistent programming model, enabling the development of real-time parallel software independent of the target architecture. The RTMOS defines the core functionality required to demonstrate the programming model.

The RTMOS functional requirements are specified using Structured Analysis and Design Technique (SADT). A hybrid of the Design Approach for Real-Time Software (DARTS) is used to perform the prelimary and detailed designs. The preliminary design is architecture-independent; the detailed design phase maps the design to a specific parallel system, the Intel iPSC/2 hypercube. The modular RTMOS design partitions operating system operations and data structures from hardware-dependent functions for portability.

The design is analysed to determine the suitability of the approach for real-time parallel systems and the portability of the design to other target parallel systems. The efficiency of the programming model and performance issues are considered.

AFIT/GCE/ENG/91M-04

A COMMON INTERFACE REAL-TIME MULTIPROCESSOR OPERATING SYSTEM FOR EMBEDDED SYSTEMS

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Michael S. Rottman, B.S.

Captain, USAF

March 1991

Approved for public release; distribution unlimited

Acknowledgments

Despite the fact that only one name shows up on the front cover, a project of this magnitude is really a group effort. One person does the actual research and writing (and takes all the credit!), but many other people contribute countless hours of advice, motivation, and support. I'd like to take this opportunity to thank a few of those people.

I am deeply indebted to my thesis advisor, Dr. Thomas Hartrum, for his technical expertise, advice, and seemingly endless patience. He deserves much credit for this thesis finally being completed and more thanks than I can express. Thanks also go to the other members of my thesis committee: Dr. Gary Lamont and Major Jim Howatt. Dr. Lamont provided valuable advice throughout this thesis and taught me that "software only exists in its documentation." If so, this must be one heck of a software design! Major Howatt's willingness to join my committee at the last minute and review this document is greatly appreciated.

I can't give enough thanks to my wife, Nancy, who gave me support, motivation, and love when anyone who loved me less would have shot me. She understood that sometimes I needed to be pushed, and other times I had to muddle through by myself ...and she knew the difference between those times. Special thanks and love also go to our new son, Kevin Patrick: though he was often a distraction, he also gave me the motivation to finally finish this project.

Finally, I would like to thank my co-workers, for giving me a great deal of support, advice I didn't always ask for, and less abuse than I probably deserved.

Michael S. Rottman

Table of Contents

The thesis design documents listed below are contained in a separate volume. This volume can be obtained by contacting Dr Thomas Hartrum, AFIT/ENG, or Mr Rudy Beavin, WL/FIGL, Wright-Patterson AFB, OH.

List of Figures

List of Tables

AFIT/GCE/ENG/91M-04

Abstract

Large real-time applications such as aerospace avionics systems, battle management, and factory automation place many demands and constraints on the computing system not found in other applications. Software development is hindered by software dependence on the computer architecture and the lack of portability between systems. This thesis specifies and designs a real-time multiprocessor operating system (RTMOS) that implements a consistent programming model, enabling the development of real-time parallel software independent of the target architecture. The RTMOS defines the core functionality required to demonstrate the programming model.

The RTMOS functional requirements are specified using Structured Analysis and Design Technique (SADT). A hybrid of the Design Approach for Real-Time Software (DARTS) is used to perform the prelimary and detailed designs. The preliminary design is architecture-independent; the detailed design phase maps the design to a specific parallel system, the Intel iPSC/2 hypercube. The modular RTMOS design partitions operating system operations and data structures from hardware-dependent functions for portability.

The design is analysed to determine the suitability of the approach for real-time parallel systems and the portability of the design to other target parallel systems. The efficiency of the programming model and performance issues are considered.

A COMMON INTERFACE REAL-TIME MULTIPROCESSOR OPERATING SYSTEM FOR EMBEDDED SYSTEMS

I. Introduction

The demands imposed by new applications continually outgrow the capabilities of existing computer systems. Every increase in computing power simply introduces new possibilities. This is especially true of embedded systems, such as flight control/vehicle management systems of future aircraft, which often operate under severe size and weight constraints. Current research into active and adaptive control, control system reconfiguration, integrated control, and artificial intelligence will place even greater demands on the aircraft control system. Future aircraft vehicle management systems will need to provide increased processing power, flexibility, and expandability. Much work remains in the areas of computer architecture and software if these demands are to be met.

1.1 The Advanced Multiprocessor Control Architecture Development Program

The Flight Control Division of the Air Force Wright Laboratory (WL/FIGL) has been investigating the application of fault tolerant multiprocessor architectures and software to flight control and vehicle management systems since the early 1980s. This work began with the Continuously Reconfiguring Multi-Microprocessor Flight Control System (CRMMFCS) in-house project, which examined the benefits of applying low cost microprocessors and VLSI technology to flight control. The Advanced Multiprocessor Control Architecture Development (AMCAD) program has continued the CRMMFCS research, defining an architecture based on non-dedicated (software-mapped) redundancy, pooled sparing, and parallel processing. A primary component of the AMCAD work is the development of the Real-Time Multiprocessor Operating System (RTMOS). The RTMOS was designed to hide specifics of the architecture from the application programmer, providing a clean, straightforward interface between the RTMOS and the application.

To provide a context for this thesis investigation, this section presents a brief overview of the AMCAD project. The first part of this section outlines several of the factors driving the AMCAD research. The next two subsections describe the AMCAD hardware architecture and RTMOS. Following that is an introduction to the AMCAD Application Interface. Finally, some limitations of the AMCAD RTMOS and Application Interface will be discussed. The interested reader can find more detailed descriptions of the AMCAD research in (43, 64, 82, 83, 85, 86).

1.1.1 Motivation The primary objective of the AMCAD research is to meet the computational requirements of future aircraft and spacecraft flight/vehicle control systems, while still providing or exceeding the required level of reliability. In addition, the research aims to reduce required maintenance through graceful degradation and continued operation with known failures. To achieve these goals, AMCAD breaks from conventional quad and triplex redundant architectures and uses instead a fault tolerant multiprocessor configuration featuring non-dedicated redundancy, pooled sparing, and parallel processing (82, 86).

Another driver of this research has been the recent push in government and industry for standardization of avionics systems hardware and software. One clear trend is towards the use of general purpose microprocessors to form common modules (85). These processors will also have to be compatible with the High Order Language standard for embedded systems: Ada. Another set of standards receiving a great deal of attention in recent years are those for data busses. High-speed data bus technology promises to replace bulky dedicated analog cabling and provide increased reliability (85). The AMCAD architecture must be adaptable to these emerging standards.

1.1.2 Architecture Overview The AMCAD architecture is based on the *module*, which consists of multiple processors (microprocessor-memory pairs) and a bus interface unit (BIU). Each processor has its own BIU access port. The processors themselves are computationally independent, but have the ability to "shutdown" the other processors by isolating them from the BIU (and therefore the rest of the system). If system consensus determines that a processor has failed, it is removed by the other processors in its module.

The modules are connected by a linear token passing bus network. This bus network is multi-level (or hierarchical) for expandability and fault containment partitioning. As a single bus is insufficient for fault tolerant operation, AMCAD places multiple busses per level.

None of the processors is physically dedicated to redundant computation; they serve instead as a pool of computational resources which may be used for parallel processing or redundant computation if needed. Redundant "channels" of computation can be software mapped across the multiple processors to provide reliability in the presence of hardware faults. This concept is called non-dedicated redundancy, and is described in detail in (85). Like the processors, the multiple busses per level are not physically redundant: communications can be distributed across the multiple busses, providing load balancing and graceful degradation if a bus fails.

One key concept is the Virtual Common Memory (VCM), which places identical copies of "shared memory" at each processor. Transmissions are broadcast to all processors, ensuring that each processor has a accurate copy of the shared memory. Write operations are thus "remote," but read operations are local, providing fast access to data and eliminating the shared memory bottleneck. Data in the VCM is protected from contamination by a segmentation scheme in which the proper key value is required for write access to a segment. This approach allows communications to occur task-to-task through the VCM rather than processor-to-processor.

1.1.3 The Real-Time Multiprocessor Operating System The development of the AMCAD RTMOS has focused on the application of real-time operating systems techniques to multiprocessor systems. Single processor multi-tasking operating systems have been used extensively for real-time control and can provide efficient and timely scheduling. AMCAD extends this model to multiple processors by statically distributing the total application workload onto the available healthy processors, giving each processor its own set of tasks to execute (64). Each processor has an identical multi-tasking kernel to manage the concurrent execution of that processor's workload. A set of upper level system functions built on top of the kernel handles the interaction of the multiple processors.

The kernels use a dynamic algorithm to schedule tasks: at any given time, the highest priority task allocated to that processor that is ready to execute is scheduled. A special timer queue is provided to allow periodic tasks to "go to sleep" until their next execution period. This multi-tasking nature of the kernels makes it possible to divide the application workload onto any number of processors; even one, if timing constraints can still be met.

Because of the high reliability required for flight control and the high threat environment in which these systems operate, the ability to redistribute the task load when a processor fails is essential (64). This redistribution of tasks, called reconfiguration, reestablishes full levels of redundancy for fault masking and provides for graceful degradation of application functions as resources are lost. When reconfiguration is triggered, each remaining healthy processor selects a new task set. This new allocation of tasks to processors will remain static until the state of the system changes again.

Intertask communication in AMCAD is based on a shared memory model. Every data item passed between tasks is assigned a "mailbox" in shared memory (the VCM). Tasks place produced data in the appropriate mailbox; tasks which need that data item remove it from the mailbox. A producer/consumer algorithm protects the integrity of this data exchange. This communications scheme is asynchronous: a producer task does not have to wait until the consumer is ready to send a data message to the VCM, though consumer tasks must wait until data is available before removing it. One advantage to using this shared memory model is that a task needs no knowledge of other tasks or processors. All a task needs to know is which data items it produces and consumes.

1.1.4 The AMCAD Application Interface A primary goal in the development of the AMCAD RTMOS was to make the specific physical configuration of the architecture transparent to the applications programmer. To this end, a clean, architecture-independent interface between the operating system and the application was developed to allow design and implementation of applications software without specific knowledge of the underlying hardware.

The approach taken in AMCAD was to present the programmer with an abstract or "virtual" machine. To the software engineer, the architecture appears as a pool of

homogenous computing elements. Each processor has an identical copy of the operating system, application software, and global data (through the VCM). Any software job can run on any processor, and the multi-tasking kernels allow the total workload to be distributed onto any number of processors. Tasks communicate through the mailboxes in the VCM, allowing tasks to operate without specific knowledge of any other task. In effect, all tasks are "remote" from one another.

The above approach forms a basic programming model or application interface which supports the development of application software *independent* of the target architecture. This model forms the basis for an architecture-independent methodology for the development of multiprocessor software for real-time systems. In addition, the portability and maintainability of the application software will be enhanced through use of this abstract interface. The RTMOS maps this application interface to the physical AMCAD configuration.

1.1.5 Limitations The application interface described above was defined as part of the AMCAD architecture and RTMOS development. This interface shows great potential for increasing the portability of application code and providing a more standard real-time multiprocessor software development methodology independent of the target architecture. However, the AMCAD RTMOS was designed specifically for the AMCAD prototype hardware and is "optimized" for that architecture. The RTMOS is also coded in assembly language. As a result, the AMCAD application interface and programming model cannot be easily transitioned to other multiprocessor architectures.

WL/FIGL would like to see the application interface developed in the AMCAD program transitioned to a better documented, more complete RTMOS specification and design suitable for implementation on different architectures. Such an RTMOS would provide a means of mapping the AMCAD application interface onto a variety of multiprocessor architectures for evaluation and experimentation.

1.2 Thesis Objectives

This thesis effort has four specific objectives. The primary objective is the specification and design of a real-time multiprocessor operating system which implements the AMCAD application interface. This programming model presents an abstract, "virtual machine" to the programmer, allowing the development of applications software independent of the particular target architecture. Building this abstract interface into a modular, well documented RTMOS specification provides an operating system design which can be mapped to a variety of parallel processing architectures.

The second objective involves mapping this common interface RTMOS design to a specific multiprocessor system. A suitable architecture is selected from those available in the AFIT environment. Once the target machine has been identified, the RTMOS design will be further refined to develop a mapping of the RTMOS to the target system.

The third objective of this thesis effort is to examine the feasibility of the AMCAD application interface for different classes of real-time multiprocessor architectures. This is accomplished by examining how the RTMOS application interface can be mapped to parallel architectures other than the the target system, the efficiency of these mappings in terms of processing overhead, and the programmability of the interface.

The final objective is to identify extensions and enhancements to the work done in this thesis. An effort of such size and scope as this cannot be be completed in the course of one thesis project and still give adequate consideration to all possible alternatives. It is the desire of the author that this thesis lead to continued efforts in this challenging area. For this reason, areas for future, related research are presented.

1.3 Thesis Overview

The development of the common interface AFIT Real-Time Multiprocessor Operating System, called ARTMOS, will follow the traditional waterfall model of the software development life-cycle outlined in (8:36) and shown in Figure 1.1. This thesis is organized around this model. Chapter 2 contains background material which serves as a foundation

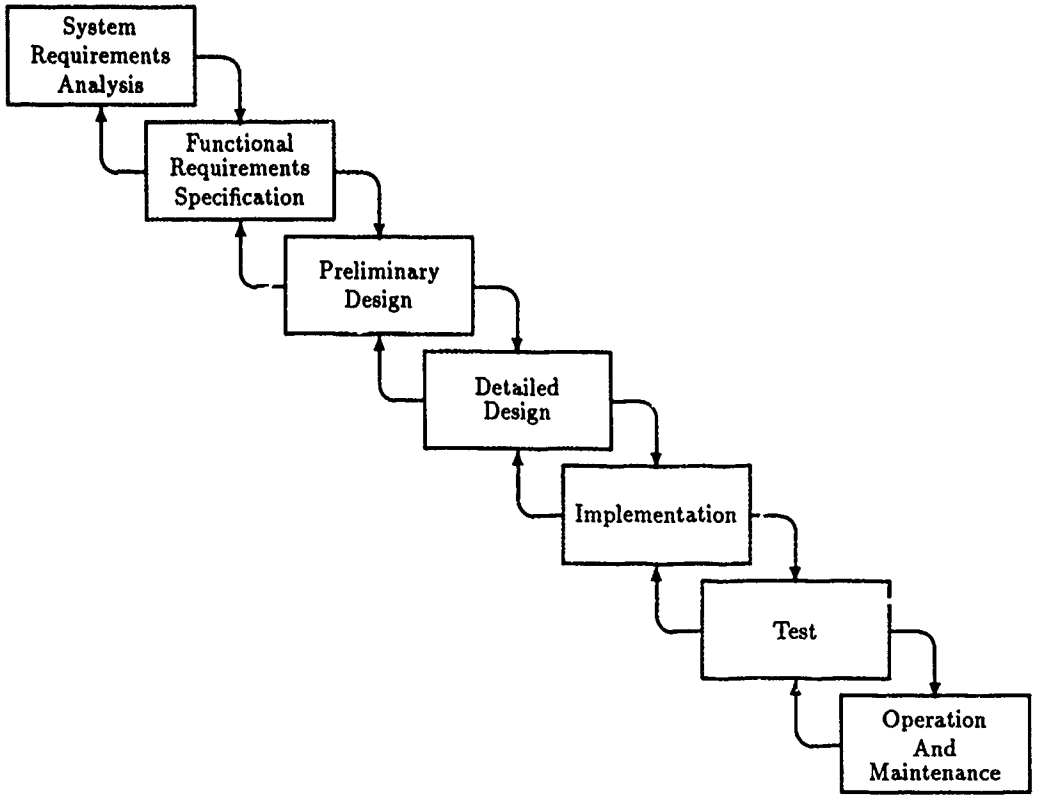


Figure 1.1. Software Development Life Cycle

for understanding this thesis. It discusses real-time systems, parallel processing concepts, and operating systems as they apply to this research.

Chapter 3 defines the system level requirements for the ARTMOS and the Application Interface. These requirements are used to develop the software requirements specification, described in Chapter 4. The software specification defines at a functional level what the ARTMOS must do without concern for how these functions are accomplished.

Chapter 5 covers the ARTMOS preliminary design. The preliminary design translates the specification into a structure which describes at a high level how the operating system will perform the specified functions. The detailed design, in Chapter 6, "fills in the gaps," defining at a lower level how the structure can be implemented to more completely show how the ARTMOS meets the requirements. Development of the preliminary design requires no specific knowledge of the target architecture. The detailed design process, on the other hand, maps the preliminary design to a particular hardware architecture.

Chapter 7 analyses the results of this research with respect to the original objectives. It considers how the application interface might be mapped to other multiprocessor architectures. It also addresses the efficiency and viability of this interface for these different architectures. Chapter 8 presents conclusions and recommendations for further work in this area.

II. Background Research

This thesis encompasses three distinct areas of research: real-time systems, parallel processing, and operating systems. The purpose of this chapter is to introduce the reader to these three topics as they relate to this particular research project. The first section deals with real-time systems, the general application area of this effort. It presents basic definitions and characteristics of real-time systems, followed by an examination of one real-time application of interest: digital flight control. Section two is an overview of parallel processing, leading into a discussion of some of the benefits and difficulties of applying parallel processing to real-time systems. The final section examines operating system concepts in the context of real-time and multiprocessor systems.

2.1 Real-time Systems

Real-time computing is a wide-spectrum research area. Large, complex real-time applications such as battle management, aircraft and space vehicle avionics systems, and factory automation (52, 92) place many demands and constraints on the computing system not found in other applications. This thesis effort focuses on this challenging arena.

This section is divided into three subsections. The first provides a general definition of real-time and introduces some real-time applications. The next subsection outlines some characteristics of real-time systems. Finally, one class of real-time system, digital flight control, is considered in more detail by looking at some of the associated requirements and future trends.

2.1.1 What is Real-Time Computing? Real-time computing differs from other types of computing in the importance of *time*. Real-time systems control their environment by receiving data, processing it, and returning the results to affect the functioning of the environment at the specified time (97:16). It is this physical impact of the system's activities upon its environment that leads to timing constraints in the form of fixed deadlines not found in non-real-time systems. Real-time systems, both the hardware and the software, must be designed to ensure that these deadlines are met. "Correctness depends not only

on the logical results of the computation, but also on the time at which the results are produced" (75:15). A right answer which is not available when it is needed is wrong, and can cause losses ranging from expensive to catastrophic. In other types of computing, such as batch or interactive processing, success depends on throughput. Tasks are performed as fast as possible, but are not constrained to complete by a certain time (39:55).

Perhaps the most important factor in the design of a real-time system is *predictability.* The functional and timing behavior of the system must be as deterministic as necessary to meet all system specifications (75:29). Predictability means it should be possible to calculate the elapsed time between an external event and the system's response, or determine with certainty the completion time of an activated task (74, 94). High system throughput may be helpful in meeting stringent timing specifications, but speed alone does not ensure predictability. "A system with unpredictable delays can never be fast enough" to guarantee that deadlines are met (94:90).

Some examples of current real-time applications include transaction processing, process control, robotics command and control systems, aircraft flight control and vehicle management systems, and space shuttle avionics systems. Space-based defense systems such as the Strategic Defense Initiative are further examples of real-time computing systems.

2.1.2 Characteristics of Real-Time Systems A real-time system is characterized mainly by the particular application for which the system is intended. A robot on a factory floor has a very different set of requirements than does an aircraft control system, yet both are real-time systems. This section outlines a number of characteristics which define the nature of real-time systems. Examples are provided where appropriate to show how specific applications affect the characterization of the system.

Throughput and Response Time. Two performance metrics often associated with a real-time system are the throughput and response time necessary to meet the deadlines imposed by the system's environment. Throughput is a measure of the computational capability of the system which describes, basically, how much work the system can do in a specified amount of time. Response time is the time it takes the system to recognize and

respond to an external event. High throughput and fast response time can be conflicting requirements since the heavier the computational load, the harder it is for the system to respond quickly and meet all deadlines. The system designer must ensure that the system has sufficient throughput to meet all deadlines despite peak workloads. The urgency of these deadlines varies from system to system as each application has its own performance requirements. A transaction processing system might take a half-second to respond to a query, which can be considered real-time for that application. Avionics systems typically require response times less than a half-millisecond (66:31).

Criticality and Fault Tolerance. The degree of fault tolerance required of a real-time system is determined by the criticality of the application. Failure of a robot in an automated factory can cause schedules to slip or serious damage, depending upon what the robot was doing when it failed, but failures in an aircraft flight control system can result in loss of both pilot and aircraft. For this reason, flight control systems typically require a probability of failure of 10^{-9} per flight hour (89:133). According to Stankovic (75:23), "failure can result from massive loss of components (static failure) or from failure to respond fast enough to environmental stimuli (dynamic failure)." Fault tolerance must be designed into real-time systems for high criticality applications to ensure that the systems meet functional and timing constraints despite changes in the status of the system.

Availability. Another consideration in the design of real-time systems is the required *availability* of the system. How long must the system sustain operation? Space systems may need to function correctly for years without maintenance. Is degraded operation allowed? What is the level of interaction with the environment? Many real-time systems are embedded in larger systems and do not appear to the user as a separate computer (65:5). Some systems need to be available to multiple users. A distinction must be made between the interval of time over which the system is expected to operate and the probability that the system is operating at a given instant in that interval. A space system may be able to tolerate five minutes of downtime per year ... unless that five minutes is during reentry!

Variability. A real-time system may have to satisfy different requirements and different timing constraints at different times during its life cycle. In an automated factory, for example, changes in the product line or new methods of production may require mod-

ifications to the factory systems. An aircraft flight control system may have more (or less) urgent deadlines for different operating modes or experience changes in its mission scenario. In either case, the application has variability which cannot be ignored by the system designer. Will the software need to be rewritten if more processors are added? Will modifications be made dynamically (during operation) or statically (prior to system use or between different uses)? The automated factory may require dynamic changes to minimize the disruption to production (75:31). Aircraft systems typically switch between different operating modes during a flight, but the systems themselves are only modified between missions.

Correctness/Validation. Correct operation is essential for any system, but correctness can be even more critical in real-time systems. Verification is the process of guaranteeing a system is correct; that is, free from faults. Verification is practically impossible for all but the simplest systems as it requires testing whether the system responds correctly for all possible inputs or mathematically proving its correctness. A program accepting real number inputs, for example, cannot be verified because every possible input value cannot be tested. Validation, on the other hand, seeks to show that a system is *reasonably* free of faults and is accomplished by testing boundary conditions and specific values of interest. Because verification is "impossible," a real-time system may need to be able to tolerate arbitrary or generic faults caused by design errors not found in testing. "Either the system must accommodate design errors in the hardware or software, or the hardware and software must both be proven correct to within the system reliability specification" (89:133).

Physical Constraints. Embedded real-time systems, such as flight control systems, are often physically constrained by their environment. Only a limited amount of space or power may be available to the system. Weight and ruggedness constraints also affect the design of these systems (99:38). In some cases, portions of the system may need to be physically distributed to provide damage tolerance.

2.1.3 Digital Flight Control Systems In recent years, the analog control systems of civilian and military aircraft have given way to digital flight control systems. These digital systems weigh less than their predecessors, use less power, and are more reliable. McDon-

nell Douglas, for example, reported a mean flight time between failures of 4.2 hours for digital avionics on the F-15, compared to 1.9 hours for analog avionics on the F-4E (73:53). Advancements in the density and speed of VLSI components allows further reduction in size of digital computers. The promise of digital control systems stems from increased integration, allowing more functions in less space, and increased reliability through redundancy of components and systems. Analog equipment also falls short of the computational flexibility of digital hardware made possible by its software (73:53).

The push towards digital systems has been driven in part by the demanding performance and reliability requirements of flight control systems. Flight control computers must provide increased reliability, high throughput, better maintainability, and reduced size, weight, and cost. The emphasis on reliability comes from the Federal Aviation Administration, which requires that flight-safety-critical controls have a probability of failure of less than 10^{-9} per flight hour for commercial aircraft (73:51). To meet this reliability level, the system must be capable of handling arbitrary fault modes.

Flight control consists of control laws which must be computed at periodic intervals based on the control mode of the aircraft, pilot inputs, and inputs from sensors and controllers. The control laws typically must execute at frequencies (iteration rates) of 40-200 Hz; that is, the control laws must be updated at least every 5 milliseconds (41:398). Trends in flight control are leading towards more control modes, increased computation in the modes, and faster cycle times. Not all modes will require the same iteration rate; there may be many different iteration rates needed by different portions of the control laws. Digital control systems for large process control plants such as flight control "require a complex real-time operating system due to the numerous individual control and sequencing operations and their interactions" (32:69). Because flight control is life critical, the system must perform deterministically in order to guarantee that timing constraints will be met.

The Air Force's F-16 C and D are the first military aircraft in production with digital fly-by-wire flight-critical controls (73:55). The four analog control computers of the earlier models are being replaced with quadruple-redundant digital computers. A multiplex bus running the length of the aircraft links the computers, sensors, accelerometers, and

servomoter that move the aircraft's control surfaces. The new system fits in a single line-replaceable unit, approximately 0.03 cubic meters and 22.5 kilograms. Perhaps the main benefit of the new flight control system is that "advanced capabilities can be added later, such as integrated fire and flight control, terrain following and avoidance, and automated management of the aircraft systems" (73:55).

The role of the flight control system for future aircraft is expanding to include many advanced capabilities and functions. Such new functions as integrated and active controls, self-repairing control system technology, and terrain following/terrain avoidance have implications for flight criticality. This focus on flight criticality introduces the concept of the Vehicle Management System (VMS). The PAVE PILLAR Avionics Systems Architecture (ASA) Specification (78) partitions the avionics suite into a VMS, which consists of all flight/life critical subsystems, and a Mission Management System (MMS), subsystems that are mission critical but not safety of flight critical. The VMS contains such functions as flight control, propulsion control, inlet control, electrical power control, and utility subsystems. Integrated control modes such as fuel management require cooperative actions of two or more of the flight critical subsystems, and may need to interact with MMS subsystems as well. The increased functionality of the VMS, coupled with additional timing constraints introduced by the complex interfaces between subsystems, drastically increases the computational requirements of the system. The VMS of future aircraft will need far more processing power than today's flight control computers, while maintaining the system reliability and decreasing the required maintenance.

2.2 Parallel Processing Concepts

The strict deadlines and high reliability required of many real-time applications often exceed the capabilities of single processor systems. Advanced aircraft avionics systems and future systems such as the Strategic Defense Initiative and Space Station have tremendous processing requirements which are driving real-time system designers towards new and more powerful processing architectures. One promising solution applies parallel processing techniques to real-time system design as a means of providing the needed throughput and fault tolerance.

This section provides an overview of parallel processing concepts; it does not attempt to fully describe any particular parallel architecture. An in-depth discussion of the myriad parallel architectures being studied in academia or available commercially is beyond the scope of this thesis investigation. The RTMOS and application interface specification and design in this research is intended to be applied to parallel systems in general, not one particular architecture.

The first subsection provides some basic definitions and terminology which are used throughout the remainder of this thesis. The next subsection outlines the potential benefits of applying parallel processing technology to real-time systems. The last subsection describes some of the difficulties of parallel processing in real-time.

2.2.1 Definitions Huang and Briggs (34:6) define parallel processing as "an efficient form of information processing which emphasizes the exploitation of concurrent events in the computing process." By their definition, parallel processing demands concurrent execution of multiple jobs or programs in the computer. This definition can be further refined to distinguish between *multiprocessing* and *parallel processing.* Multiprocessing uses multiple processing elements to run totally different applications or independent executions of the same application in order to increase the number of jobs or programs processed per unit time (17:19). This approach typically does not reduce the time to execute a single program (42:91). Parallel processing, on the other hand, applies the multiple processors to solve *one problem* or *application* (42, 17, 59, 34, 77). Simultaneous execution of different portions of a problem reduces the "runtime" of the problem.

Parallel processing, then, uses two or more interconnected processors to simultaneously perform different portions of a single application. An application may be simple, consisting of an individual function or algorithm to be parallelized, or a complex collection of related programs which together solve a large computational problem. Quinn (59) deals with the former, presenting a detailed discussion of the development and implementation of parallel algorithms. Sorts, graph searches, matrix operations, and fast Fourier transforms are examples of single function applications. Real-time systems, however, typically consist of a number of interdependent tasks performing functions related to the overall completion

of a large, complex application (31:66). An aircraft flight control system, for example, might include as *sub*-functions a Kalman filter algorithm, a fast Fourier transform, or control matrix operations, whose parallelization must be based on the overall application requirements rather than just the "best" parallel implementations of the algorithms themselves.

Related to parallel processing is distributed processing, which disperses the computing functions among several physical computing elements which may be co-located or geographically separated (45:3). A distributed system may or may not be a parallel system, depending upon the level of cooperation between the computing elements. A network of independent computers working on independent problems is distributed processing, but not parallel processing because the computers are not cooperating to solve the same problem. Liebowitz and Carson (45:5–6) use two criteria to classify distributed systems: geographical separation of processors and degree of cooperation (interaction) between the processors. A locally distributed multiple processor system, for example, consists of physically independent processors linked by an interprocessor communications system. The processors are local to one another, within several hundred yards, so the communications links can have high bandwidth and reliability yet still be relatively inexpensive. The processors cooperate closely to solve a single problem or related group of problems.

Parallel processing can be either internal or external. Internal parallelism focuses on enhancing the concurrency of single processor systems, typically through the use of pipelines. Most computers and even microprocessors have some form of instruction pipelining that lets instructions overlap, thus speeding throughput (10:52). Arithmetic operations can be segmented so that each stage of the pipeline performs a specific step of the task, then passes the results to the next stage. At a higher level of parallelism, multiple functional units, such as adders or multipliers, can be incorporated within a single processor to allow concurrent execution of instructions (17:18). Another type of internal parallelism, vector processing, repeats the same instruction for a list, or vector, of data. Some machines are capable of processing the entire vector simultaneously. Vector processors, however, are efficient only if the arithmetic functions to be performed can be arranged as continuous streams of data (10:52).

In contrast, external or global parallelism seeks to improve performance by dividing up a problem and processing its parts simultaneously on multiple processors. The individual processors can range from extremely simple arithmetic logic units to supercomputer-level processing units, themselves incorporating some measure of internal parallelism. Multiple processor architectures are classified as single-instruction-stream, multiple-data-stream (SIMD) or multiple-instruction-stream, multiple-data-stream (MIMD).

In SIMD architectures, a single program is used to operate on many data simultaneously (77:279). Multiple synchronized processors are supervised by the same control unit. Each of these processors executes the same instruction broadcast from the control unit, but operates on different data elements (34:34). Through the use of masking, some of the processors can be programmed to ignore certain instructions. "This ability to mask processing elements allows synchronization to be maintained through the various paths of control structures, such as clauses of an **if ... then ... else** statement" (59:30).

MIMD architectures, on the other hand, consist of multiple independent processors, each capable of executing its own program, interconnected in some manner to allow exchange of data and synchronization information (77:279). MIMD systems are typically either tightly or loosely coupled. Processors in a tightly coupled MIMD system communicate through a common memory space. Access to the shared memory is through a central interconnection network such as a shared bus, crossbar switch, or a packet-switched network. Performance in tightly coupled systems tends to degrade rapidly as the number of processors grows large due to bottlenecks in accessing the shared memory (34, 59, 54). A loosely coupled MIMD system is composed of modules, each containing a processor with local memory and input/output devices. Tasks running on different modules communicate by exchanging messages through a message-transfer system (34:460). Because the coupling between modules is so loose, these systems are often referred to as distributed systems. Hybrid systems, combining aspects of both tightly and loosely coupled system, are also possible (the AMCAD system described in Chapter 1 is one example).

Of the two types of external parallelism, MIMD systems are most suitable for real-time applications. SIMD machines are not well suited for applications which cannot be organized into repetitive operations on uniformly structured data and whose addressing

patterns are data dependent (77:279). Large, complex real-time applications require the simultaneous processing of many tasks which may be related parts of a larger function or which may be completely independent (31). MIMD architectures, composed of asynchronous and independent processors, adapt well to tasks requiring simultaneous but different operations by their very nature. Processors in an MIMD system can all perform different functions, communicating only when they require data or synchronization. Huang and Briggs note, however, that some real-time applications can benefit from SIMD operation; image processing and pattern recognition, real-time scene analysis, and automated map generation are examples (34:397–398). In these cases, a SIMD array processor might be needed as a co-processor.

This terminology is by no means universally accepted. Parallel processing classifications are loose, overlapping, and subject to much debate. Definitions tend to vary greatly between references for this very reason. To many researchers, for example, a multiprocessor architecture consists of multiple processors in a tightly or loosely coupled configuration. Others claim that a system is a multiprocessor only if it includes shared memory. For this thesis effort, the term *multiprocessor* refers to both tightly and loosely coupled MIMD systems which are either centralized or locally distributed. The term *parallel processing* refers to the concurrent processing of many related programs to solve large, complex real-time and embedded systems rather than the splitting of one particular algorithm across multiple machines.

2.2.2 Benefits of Parallel Processing A primary objective of parallel processing is to increase the performance and throughput of the computing system. Many applications require processing speed beyond the ability of a single processor to provide. Large, complex real-time systems, because of their stringent timing constraints and diverse workloads, place even greater demands on the computer. Because the speed of individual processors is physically limited by existing device technology, "the motivation for moving towards multiple processors is strictly a matter of performance" (77:271). By using a number of processors, an application can be divided into parts which may be run in parallel to reduce the time to perform the calculation. "Parallel implementation can achieve

orders-of-magnitude faster execution time (depending on the number of processors) than uniprocessor implementations" (31:66).

Parallel systems can also provide higher levels of reliability and availability. Embedded systems such as aircraft flight control and avionics systems must continue to operate correctly despite faults or battle damage. Space systems must function for extended periods of time without maintenance. Parallel processing is one means of achieving the required reliability and availability. The basic idea is that if any processor fails, its workload can be performed by other processors in the system (77:281). Such a failure may result in a decrease in performance as a processor is lost, but the system can continue to function correctly. The parallel system can provide high reliability and availability in two ways (20, 42): reconfigurability and redundancy. Reconfigurable parallel systems spread different portions of the workload across independent processors to allow graceful degradation as processors fail by redistributing the workload to the remaining healthy processors. Redundant computation, on the other hand, has multiple processors working the same problem and data and voting the results to provide fault detection and masking.

Another advantage of parallel processing is expandability. Over the lifetime of a complex real-time system, particularly in military systems, requirements tend to evolve as new functions are added or missions change. If the capabilities of a single processor are exceeded, the entire system must be replaced with one able to meet the new requirements. With parallel machines, more processors can be added (within limits) to keep pace with applications that grow in size or performance requirements (42:91).

Parallel processing can decrease the cost of high performance over single processor systems. In order to achieve increased power, single processor systems must constantly incorporate faster and more expensive circuitry. These higher-speed processors are reaching physical limitations based on the maximum velocity electrical signals can propagate and the ability of the circuits to dissipate the generated heat (17:16),(20). By harnessing many relatively inexpensive, slower-speed processors together into a multiprocessor system, the cost of high performance can be significantly reduced (77:282). In general, parallel processors tend to have better price-to-performance ratios than sequential architectures (42:91).

The extra computing power afforded by parallel processing can also be used to improve the performance or functionality of real-time applications. Increased processing speed can allow the use of higher frequency control rates or more complex control laws. Spare computing time can permit the incorporation of additional functions and features which a single processor system might not be able to handle. Also, the extra computing power might enable the use of higher order languages, a standard operating system interface, or a more straightforward software implementation without causing deadlines to be missed (91:2).

2.2.3 Challenges The advantages must be balanced by consideration of some of the problems associated with parallel processing. Stone (77:281) and Haynes et al. (30:88) discuss a number of factors which can limit the efficiency of parallel processing. One such factor which can reduce performance is delay for interprocessor communications, whether it be processing overhead for message passing or contention for shared memory. Synchronizing the work of one processor with that of another can lead to further delays. Another limiting factor can be the overhead required by the operating software for task scheduling and system control. Task scheduling and synchronization operations can degrade multiprocessor performance to a much greater extent than in single processor systems (77:281). Contention of the multiple processors for shared variables or system resources can slow the execution of individual processors. In addition, performance can be reduced if the application workload is not balanced across the processors, causing some processors to sit idle while others have more work than they can handle.

A final source of inefficiency can stem from the particular algorithms of interest. Some algorithms have little inherent parallelism, making them difficult to implement efficiently on parallel machines. Likewise, some algorithms will perform poorly on certain architectures (40:44). Stone points out that "when parallelism cannot be tapped effectively, it simply adds to the cost and complexity of the system" (77:283).

Exploiting the parallelism of a given problem can be an extremely complicated process. One approach employs parallelizing compilers to identify and take advantage of the implicit parallelism of sequential programs. The problem with this approach is that cur-

rent compiler technology only identifies instruction level parallelism and is unable to alter the underlying algorithm to achieve more parallelism (19:52). A second approach is to explicitly define the parallelism in a program using operating system or language constructs. Difficulties to this second approach include synchronization, efficiency, debugging, and portability (19:52). These difficulties are exacerbated in real-time systems, with their demanding speed, determinism, and reliability requirements. The key issue here is that development and implementation of parallel programs is *not* a trivial process.

In addition, the designer of parallel software must contend with different communications mechanisms and processor interconnect schemes for each target machine. Every architecture forces different constraints and implementation details onto the programmer which can significantly impact the software design. Because parallel software is so closely linked to the target architecture, no truly standard software development methodology is available to assist the programmer in designing software for parallel computers. "If multiprocessors are ever to become a useful, general-purpose tool, they must present an abstract view of the underlying hardware to the user ...a programmer should not be expected to tailor an algorithm to suit the underlying hardware" (79:141).

Proper design and assignment techniques are necessary to address some of the inefficiency and load-balancing issues. One possible solution to the parallel design issues might be to define a common application program interface for the operating system. Such an interface could reduce the complexity and variability of parallel software design, permitting a more standard design methodology to be practiced.

2.3 Operating Systems

This thesis effort focuses on operating system concepts for the demanding application areas of real-time embedded systems and parallel processing. The purpose of this section is to examine operating systems in the context of these two application areas, followed by consideration of some of the challenges to developing and using an operating system for real-time parallel computers. First, a brief overview of operating systems and their functionality — what they are and what they do — is presented. The following two subsections discuss real-time and multiprocessor operating systems, respectively. The final

subsection deals with some of the issues and challenges of an operating system intended for real-time *and* multiprocessor systems.

2.3.1 Operating System Functions Quite simply, an *operating system* (OS) is system software which provides an interface between the bare hardware of a computer system and the user of the computer. The OS shields programmers from the complexity of the system hardware by supplying an efficient environment for the development and execution of user programs (56:3). From this definition, the purpose of the OS is two-fold: to make the computer system *convenient* to use, and to use the computer hardware in an *efficient* manner.

One goal of the OS is to provide the user with a convenient interface to the system. The OS shields the programmer from low-level details of the system hardware such as interrupts, timers, memory management, disk access, and other features with which the programmer typically does not need or want to deal. In effect, the OS provides the programmer with a high-level abstraction that is simpler and easier to use than the hardware. From this point of view, "the function of the operating system is to present the user with the equivalent of an extended machine or virtual machine that is easier to program than the underlying hardware" (80:4).

The second goal of the OS, which sometimes conflicts with the previous goal of convenience, is that the system hardware be used efficiently. In this view, "the job of the operating system is to provide for an orderly and controlled allocation of the processors, memories, and I/O devices among the various programs competing for them" (80:4). Because computer systems are expensive and it is desirable to make full use of their capabilities, most operating system theory focuses on the optimal use of computing resources (56:4). The operating system's role as *resource manager* involves tracking who is using which resource, granting resource requests, accounting for usage, and mediating conflicting requests from different programs and users.

The primary function of an operating system is the latter of the two roles, resource management (14:3). In order to understand how the operating system performs this function, it is necessary to first examine the available resources. Tanenbaum (80:42) identifies

four major resource management functions an operating system must perform: process management, memory management, file management, and input/output (I/O) device management.

A *process* (or *task*) is "the activity resulting from the execution of a program with its data by a sequential processor" (6:36). The state of a process is maintained in a data structure (often called a Process Descriptor (14), Process Table (80), or a Process Control Block (48)) which describes the status of all hardware and software entities the process uses. Processes operate almost independently of one another, cooperating by sending messages and synchronization signals and competing for resources (6:37). The OS provides facilities for managing the creation, scheduling, communications, synchronization, protection, and termination of these processes. Some processes may run programs in response to user commands, while others are system processes for handling file requests, disk access, and so on (80:49).

A second resource management function of an OS is memory management. The main memory, also called executable memory because a program cannot run unless it is loaded in memory, can be directly accessed by the processor as data or instructions (6:185). Memory is usually a limiting resource in a computer system because it is expensive and needed by every program, so efficient memory management is a critical aspect of system performance. The size of the main memory can affect user programs in two ways (6:185). Many applications are large enough that they exceed the amount of physical memory. Programs may need to be divided into smaller parts which can be loaded into memory as needed. Multiprogrammed systems, on the other hand, may need to keep several processes in memory at the same time. Many of the performance benefits of multiprogramming are negated if several processes cannot be kept in memory, ready to execute (56:143). Swapping techniques are used to switch processes or parts of programs in and out of memory, while paging and segmentation schemes can give the appearance of much more physical memory than is actually present ("virtual" memory) (80:191). The memory manager tracks which parts of memory are in use and which are not, allocates memory to processes when needed and deallocates it when they finish, and controls the swapping between main memory and secondary storage when needed.

Secondary or backing storage provides a means of permanent storage, allowing the user to keep information until it is needed (6:258). Programs cannot be executed out of secondary storage; information must be accessed and loaded into main memory before it can be used (56:68). Secondary storage can consist of a several different mass storage devices, such as magnetic tape, disks, or drums, each with its own characteristics and physical organization. The operating system abstracts the physical properties and peculiarities of the physical devices to present the programmer with a clean, device-independent view of information (80:16). This logical storage unit is the *file*. The file system is responsible for creating, destroying, organizing, reading, writing, and controlling access to files (6:259).

The final resource managed by the OS is input and output. Computer systems must be equipped with devices to allow the user(s) to interact with the system (6:258). Devices such as terminals allow the user to input commands and requests to the system and receive information or results back from the system. Printers are another type of output devices. As with storage devices, the OS must provide an abstract interface between I/O devices and the system that is easy to use and device-independent (80:110). The OS also provides mechanisms for the efficient sharing of physical I/O devices and the protection of data transferred or managed by the I/O devices (6:259). Most device management is spent honoring I/O requests from processes in the system (14:127).

Computer operating systems vary greatly depending on both the architecture and the application of the system. Types of operating systems include interactive, batch, multi-user, multiprogrammed, personal computer, time-sharing, real-time, and multiprocessor. Of particular interest to this thesis are real-time and multiprocessor operating systems. These types are examined in more detail in the next two subsections.

2.3.2 Real-Time Operating Systems The primary difference between a real-time operating system (RTOS) and other types of operating systems is the importance of time. "A real-time operating system by definition has to process certain tasks within specified time intervals. In contrast, a non real-time systems's main objective is to process tasks in an efficient manner (time and space) without regard to initiating task activity at a precise time instant" (32:474). More specifically, a RTOS monitors and controls external

processes with strict timing constraints on response to events. Events typically command the attention of the RTOS through use of interrupts. If interrupts are not handled promptly (based on the timing constraints for the given application), information may be lost or the external processes may be seriously degraded or misrepresented (6:22).

The event-driven nature of real-time systems drives the functionality and operation of real-time operating systems. A RTOS must be capable of responding to event signals, determining which event is occurring, processing event data, and responding to drive the state of the system as required (22:29). If multiple events occur simultaneously, the RTOS must be able to select one to operate on based on some priority criterion for the given conditions. The RTOS must also manage system resources (as must a general-purpose OS), schedule periodic tasks, coordinate intertask communications, and control system integrity (32:475).

At the heart of every real-time operating system is a real-time multitasking kernel. The kernel is essentially a minimal set of operating system facilities required to schedule tasks, manage resources, and provide mechanisms for intertask communications and synchronization (69:122). The low-level kernel functions provide the basis for the construction of the rest of the operating system. Because of the stringent timing and determinism required of real-time systems, the efficiency (time and space) of the operating system is critical. Unnecessary functions and inefficiently implemented OS primitives can result in missed deadlines or excessive memory requirements. Minimization of OS overhead for such functions as task scheduling and resource management may be essential to meet processing requirements (32:473).

Stankovic (74:4) examines some of the salient features of real-time kernels for current systems. To reduce run-time overhead and usage of precious system resources, kernels need to switch between tasks and respond to interrupts quickly, while minimizing the time during which interrupts are disabled. Memory management is limited to fixed or variable sized partitions (no virtual memory). Kernels must also provide the ability to lock code and data in memory. To deal with timing constraints, kernels maintain a real-time clock with primitives to allow tasks to delay by a fixed amount of time. Kernels typically are very small in size because of their minimal functionality, which differs from kernel to kernel.

In VxWorks, from Wind River Systems, functions such as memory allocation, timing, and I/O system calls are excluded from the kernel definition (95:2). The Intel iMRK real-time kernel includes these features and more (39).

One difference between real-time and non-real-time operating systems is the issue of fairness. Resource management in non-real-time system is fair: processes of equal priority have equal access to the processor and an equal amount of processing time if needed. In real-time systems, however, processes are assigned priorities based on the urgency of their deadlines and their importance relative to the other processes (74:4). It is essential that the highest priority process that is ready to run at a given time is actually running. "Fairness is not an appropriate abstraction of the way processes are scheduled when real-time is of concern" (75:17). Likewise, some real-time tasks and operating system routines which require extremely fast execution must be locked in memory to minimize response and execution time.

2.3.3 Multiprocessor Operating Systems Multiprocessor operating systems (MOS) are concerned with the control and management of parallel computers, extending the single-processor multitasking of computation onto multiple processors. Because of this, MOSs tend to have many features in common with multiprogrammed operating systems (34:526). Both must provide functions for task scheduling, resource allocation and management, memory and data protection, deadlock prevention, and abnormal process termination. A MOS, however, must also handle the additional complexity of the multiple processors and their interactions. Mechanisms are needed for communications and synchronization of tasks across processor boundaries, and the efficient utilization of multiple resources, such as distributing the task load evenly onto the multiple processors so that all are kept busy (load-balancing). An additional burden on the MOS is support for the exploitation of system parallelism: a poor OS will negate the advantages of parallel processing. As such, interprocess communications, synchronization mechanisms, and process placement and assignment policies dominate the efficiency of the OS (34:526).

An important consideration in multiprocessor systems is the configuration of the operating system. Huang and Briggs (34:526–528) describe three common configurations.

In a *master/slave* operating system, one processor (the master) is responsible for system management, scheduling and allocating work (processes) to the slave processors. The *separate supervisor* operating system places a copy of the operating system at each processor, allowing each processor to service its own needs. Because of the interactions between the processors, supervisor code must be reentrant or replicated at each processor. The *floating supervisor* configuration uses a single copy of the operating system in global memory. The supervisor "floats" from processor to processor as different processors access portions of the supervisor code. In this configuration, several processors can be running supervisor code simultaneously.

While there are few pure examples of these classes, they demonstrate types of centralized and decentralized system control. Centralized supervision eliminates the need for redundant code and system information and can result in more efficient processor utilization, provided the supervisor can keep up with requests for work (34:527). Because system control is centralized, however, the system is highly susceptible to supervisor failure (14:206). Decentralized supervision takes advantage of the fact that many system functions, such as handling of an I/O device error, can be done without accessing central information (14:207). Decentralization can also increase reliability by reducing the criticality of any one hardware element; if no hardware unit is critical, system performance continues in the event of a hardware failure. The cost of decentralization is increased contention for shared resources, reentrant or replicated supervisor code, and complex synchronization (34:526–528), (14:205–207).

The multiple processing elements of a multiprocessor system add to the complexity of resource management. The cost of accessing memory (in time and interprocessor contention) often depends on which processor is accessing which section of memory (14:207). As a result, the allocation of code and data can greatly affect the operating efficiency of the system. Another source of added complexity is the scheduling of processes to processors. The simplest approach is to define processes assignment statically at design or compile time (14:208). Each processor schedules and manages its set of processes. This approach, though simple, can cause poor processor utilization if the workloads are not evenly balanced. A more efficient method avoids static assignment entirely, dynamically assigning

ready processes to available processors. A given process is not constrained to execute on the same processor each time. Determination of the optimal processor-process allocation can require exponential time, however (14:208). Because processes can execute *simultaneously* on a multiprocessor, interprocess communications and synchronization are more complex than on a uniprocessor system. Mutual exclusion of shared resources becomes even more critical (14:209).

An important function of a MOS is to shield the programmer from any asymmetry in the hardware or software, whether due to the original design or a failure (14:206). This attribute is known as *transparency.* The OS must present an abstract view of the hardware to the programmer such that the programmer can request an action by specifying what is to be done and not be required to specify what physical or logical component is to provide the service (21:139). Accordingly, the programmer should be able to design application software without concern for which process executes on which processor, or the number of processors available in the system.

A final consideration, in the context of real-time systems, is the physical distribution of the processors. The reliability requirements of complex military real-time applications such as flight control may demand that processing elements be physically distributed to minimize the effects of battle damage. This distribution places additional responsibility on the OS. Time delays in the collection of system status information may result in decisions made with out-of-date information. It is also possible to have variations in the information presented to different parts of the system (21:142). Additional OS overhead may be needed to protect the integrity of data exchanges.

2.3.4 Challenges Parallel processing is a promising means of providing the performance and reliability required for real-time systems. Operating systems for these two types of systems have similar functionality, but each focuses on different issues. Selection or development of an operating system for a real-time multiprocessor system must focus on the requirements of both types of systems, posing a number of challenges.

One serious issue in real-time multiprocessor systems is process scheduling. Real-time systems are concerned with "completing all aperiodic (and periodic) activities at the

most valuable times, neither too early or too late, despite dynamic resource demands and conflicts, processing overloads, and hardware or software faults" (1:57). Scheduling algorithms in multiprocessor systems primarily emphasize load-balancing and efficient use of resources. Most of the load-balancing algorithms do not consider timing constraints of tasks and cannot be directly applied to hard real-time systems (16:165). Many multiprocessor operating systems dynamically schedule processes to run on available processors in order to maximize the utilization of the processors and for adaptability to environmental changes. Dynamic scheduling, however, involves higher run-time costs and must account for communications delays in collecting state information for scheduling decisions (16:165). Scheduling algorithms must be selected which balance these conflicting requirements.

Commercially available real-time operating systems are not equivalent. All provide the same basic functions of multitasking and intertask coordination, but no two implement these features in exactly the same way (70:186). While many of the features of the various commercial real-time operating systems appear similar, such as semaphores, mailboxes, and queues, they are not uniformly implemented. "Each real-time operating system provides a suite of primitives having subtly, but significantly, different properties" (69:120). These operating systems differ not only in the functions provided, but also in the way the OS functions are invoked: some implement a software trap while others use the subroutine mechanism of the target processor. As a result of these implementation differences, selection of a RTOS demands close and careful examination.

RTOS support for multiprocessing also varies significantly between different implementations. Most RTOS are designed for one particular type of multiprocessing, whether it be shared memory, message-passing, or some hybrid. Some support several types of multiprocessor systems and allow the user to select the appropriate communications protocols for the given architecture and application. In many cases, support for multiprocessing is designed into an existing RTOS. Such an approach may not be able to take advantage of different architectural features (67:255). Few real-time operating systems, however, present the programmer with a "virtual" or abstract model of the underlying architecture. The user must tailor the application and algorithms to the target architecture. As a result, applications software for real-time multiprocessor systems must be designed from scratch

for each situation, is not portable, and is not reusable. Also, the addition or deletion of processors or the repartitioning of applications tasks among processors should not require software redesign (74:3).

Most commercial real-time operating systems attempt to be as flexible and general-purpose as possible in order to appeal to the widest market. Any feature the user can imagine needing is available in some fashion. The Intel iRMX RTOS, for example, provides eighteen system calls for object oriented programming support alone! (36:5) While these RTOSs can typically be configured to include only essential functions, application reliance on non-standard features limits the portability of the software.

In general, the strict timing constraints of real-time systems has forced the developer of real-time software to take a minimalist, low-level approach to software design to increase performance. Software is designed to take full advantage of any hardware or operating system features which can reduce response time or increase throughput. Real-time software is often unsuitable for applications other than the one for which it was designed (1:50). This intimate relationship between application and system reduces the portability and reusability of the code, forcing real-time software developers to essentially start from scratch for each new project. "Because real-time software must operate under rigorous performance constraints, software design is often driven by the hardware as well as the software architecture, operating system characteristics as well as application requirements, programming language vagaries as well as design issues" (58:367). Parallel processing offers one possible means of meeting real-time performance requirements without forcing such intimate links between the architecture and the application.

One possible solution to some of these problems is for the RTMOS to present the programmer with a higher level of abstraction which is independent of the underlying hardware. A logical, architecture-independent programming model would allow the programmer to concentrate on designing algorithms without being forced to deal with low-level hardware details. Programs designed for this abstract or virtual machine model would be portable to other machines implementing the same model. The OS would support this abstract programming model in the OS interface and primitives, but could be designed to optimize the implementation for the particular architecture. The programming model

defines *what* functions are available; the *how* is defined by the OS implementation. Because the software would not rely on a particular hardware configuration, the number of processors could change or the workload could be redistributed without affecting the software. "A kernel interface is more than just a mechanism for accessing physical resources. It is also a programming abstraction that profoundly influences the algorithms that can be implemented on top of it" (67:256).

2.4 Chapter Summary

This chapter examined the three basic elements of this thesis project: real-time embedded systems, parallel processing, and operating systems. Large, complex real-time systems such as aircraft or spacecraft flight control/vehicle management systems have strict performance requirements that must be met regardless of system loading or failures. To ensure real-time timing constraints are met, the system must be predictable.

Parallel processing shows a great deal of promise for providing the performance and reliability needed for real-time embedded applications. By increasing the processing power of the system, applications programs can be processed more quickly and new features can be added as necessary. The additional processors allow computation to be distributed or replicated, making the system more tolerant to failures or damage.

Operating systems bridge the gap between these two areas to realize the potential of parallel processing for real-time applications. A real-time multiprocessor operating system must support the basic functions to meet real-time constraints and predictability requirements, while providing efficient management of multiprocessor system resources. Ideally, the low-level functionality needed to satisfy these requirements should be transparent to the programmer, who is concerned with designing and developing a particular application and not an operating system.

III. Requirements Analysis

The first stage of the software development life cycle discussed in Chapter 1 and shown in Figure 3.1 is to define the system requirements. The definition of requirements for a software project consists of two related activities: requirements analysis and functional requirements specification. Requirements analysis is an assessment of the needs the software is to fulfill when implemented. The product of this analysis is a complete, consistent, and unambiguous description of *what* the software should do but not *how* to do it (8:36). These requirements serve as a guide for the development of the functional requirements specification. This activity specifies the functions to be performed by the software and the data to be processed to satisfy the requirements, without describing how the functions are to be implemented. The specification not only guides the design and implementation stages, but also serves as a standard against which the final product is measured (99:36). This chapter describes the requirements analysis for the AFIT Real-Time Multiprocessor Operating System (ARTMOS). Chapter 4 develops the functional requirements specification.

The requirements analysis for the ARTMOS considers three types of requirements. The first subsection describes the functional, or application driven, requirements. These requirements define the specific functional capabilities the ARTMOS must have to satisfy the needs of real-time multiprocessor systems described in Chapter 2. The next subsection examines the performance characteristics required of the ARTMOS. The final subsection presents ARTMOS requirements derived from software engineering principles.

3.1 Guidelines

The primary objective of this research is to specify and design a real-time multiprocessor operating system based on the AMCAD application program interface (API). It may be necessary to refine or enhance the AMCAD API based on the analysis performed in this investigation. A secondary objective is to specify and design the ARTMOS in a consistent, modifiable, well-documented manner. This second goal should enhance the portability and maintainability of the ARTMOS design, as well as making it more understandable for

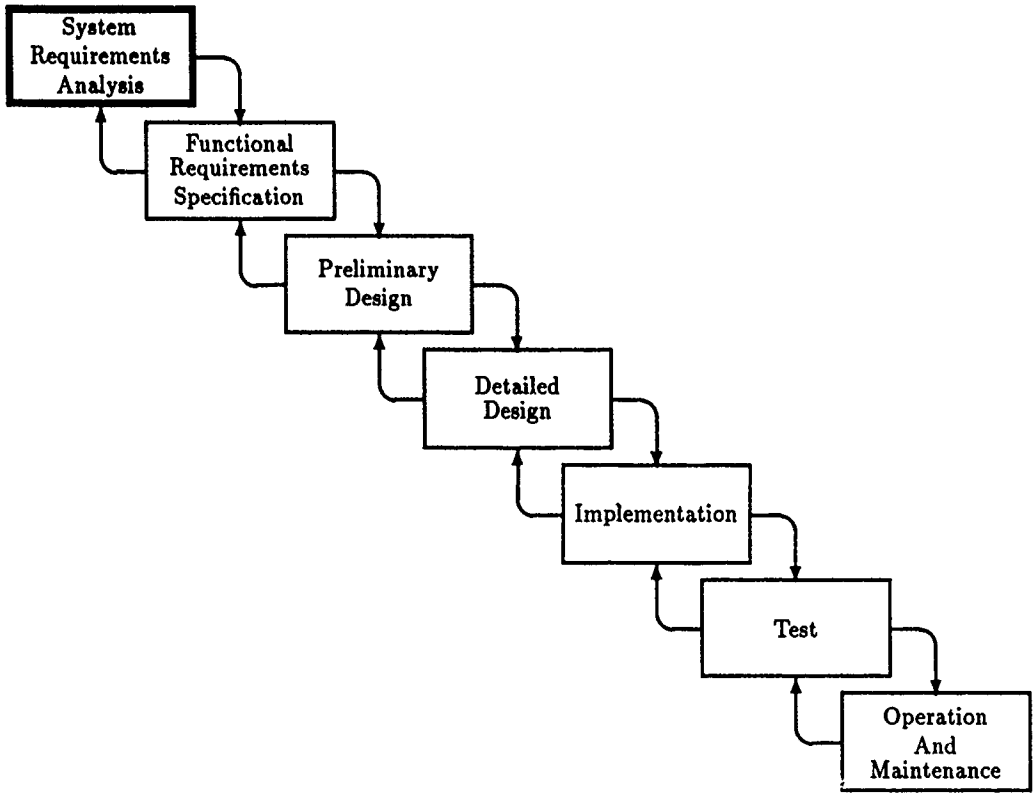


Figure 3.1. Requirements Analysis in Software Development Life Cycle

follow-on efforts. The ARTMOS is not intended to be a general purpose operating system or even a general purpose *real-time* operating system. It is being designed for large, complex real-time applications such as flight control, aerospace vehicle management, and military command and control.

Given these objectives, it is not necessary or even desirable to include all possible features that might be needed in a real-time operating system. Rather, the goal is to define only those kernel functions required to implement and demonstrate the application interface. This functionality provides a small and efficient set of primitives which may be used to build application systems. Analysis of existing real-time operating systems and the application requirements is needed to identify the minimal or core set of functions.

The efficiency (space and time) of the ARTMOS is a primary concern, and should take precedence over functionality beyond the required minimum. Since the ARTMOS is

intended as the basis for larger systems, it cannot take up space and time required by the application using it.

The ARTMOS is not intended for multiuser systems, nor is the concept of system "user" the same as found in mainframes or personal computers. For real-time systems, particularly those in physically embedded systems such as aircraft flight control/vehicle management systems, the "concept of independent users does not exist as it does in typical timesharing and computer network systems. Rather, the whole system is dedicated to performing a particular mission that can be thought of as having a single user, comprised of the physical processes being controlled" (52:9).

3.2 AMCAD Application Interface

All real-time operating systems provide the applications programmer with some form of programming abstraction. This programming model is defined by the primitive functions provided by the RTOS and the manner in which these functions influence software development. In many cases, the application interfaces provided by commercial RTOSs are unsuitable for complex, distributed real-time applications because they assume specific architectural characteristics, such as shared memory, which may not be available, or because they push details of the hardware configuration into the application software. The RTMOS developed in the AMCAD project (64) defined an application programming model intended for these critical, distributed real-time systems, such as flight control, which emphasizes hardware transparency and consistent primitive functions. The primary objective of this thesis investigation is to specify and design a portable, well documented RTMOS based on this programming interface.

A major goal of the AMCAD operating system was to make the physical hardware configuration transparent to the application programmer. To this end, a single processor multitasking kernel model was extended to multiple processors such that application software never needs to know how many processors are available or how tasks are allocated to processors. Intertask communication occurs through mailbox structures rather than processor-to-processor, so the task model is identical whether implemented on a uniproces-

sor or multiprocessor system. The programmer can focus on specifying tasks and intertask relationships independent of the hardware configuration.

Each data item passed between tasks is assigned its own unique mailbox structure. Tasks needing data request it from the data item's mailbox, while tasks producing data send it to the appropriate mailboxes. Tasks can be structured as "servers" which wait for data to arrive, perform some action or transform the data, and generate results. Producer tasks are not required to know who uses the data they produce; consumers do not care who generated the data they use. With this task model, "client" tasks *activate* "server" tasks by passing them the data they are waiting for, essentially performing *remote procedure calls* (84).

As a result, each task can be specified, coded and tested relatively independently from one another. Tasks are only required to know which data variables they use, not which other tasks they cooperate with. In essence, the AMCAD binds tasks to their data, not to other tasks.

3.3 Hardware Model

A model of how the ARTMOS views the system architecture must be developed. This hardware model describes exactly what the ARTMOS expects from the system hardware, and must be carefully defined, as many of the functional requirements are affected by the hardware model.

For the ARTMOS to be portable to other machines, how should the operating system view the hardware? This question is even more critical for parallel architectures than for single processor systems because of the greater number of possible hardware configurations in parallel systems. In a single CPU system, the software manages the processor, memory, and any peripheral devices. Many real-time operating systems, such as VRTX (35) and pSOS (71), minimize hardware dependence by managing only the processor and memory. Special system calls allow the user to initialize and use additional devices: timers, character I/O, etc. This approach is extremely flexible, allowing the user to easily configure the system for needed devices and application specific handling requirements.

A parallel system is more complex because of all the possible ways of interconnecting the multiple processors. Not only must the operating system be tailored for the particular configuration of each processor module, which may differ between modules, it must also be configured for the specific mechanism linking the modules. The interconnection schemes range from tightly-coupled, shared memory machines to hierarchical message-passing systems, with many hybrid architectures in between. Because so many parallel machines are possible, designing portable software is not a trivial task.

How, then, can the operating system be designed to encourage portability? One critical factor in the operating system design is be the architecture model used by the OS. Of the two extremes, shared memory and message-passing, message-passing is the more flexible for three reasons:

- Shared memory is a special hardware device. Dependence on specialized hardware, which may not always be available or practical, hinders portability to systems without shared memory.
- Algorithms based on message-passing can run on systems with or without shared memory, but not the converse.
- Though fault tolerance is not an explicit requirement for the ARTMOS, a physical shared memory presents a single point failure which cannot be tolerated in a flight critical system.

The ARTMOS design therefore uses a hardware model based on message-passing rather than shared memory. This ensures the flexibility necessary to transport the design among different hardware architectures.

3.4 Functional Requirements

This section defines the functionality of the ARTMOS. As described above, the ARTMOS consists of a minimal set of functions necessary to demonstrate the AMCAD API and provide the programmer with the concurrent mechanisms needed for real-time embedded software. Analysis is performed to identify the necessary functions and their manner of operation.

To form a basis for identifying the required ARTMOS functions, several real-time operating systems are examined. These operating systems are analyzed with respect to the requirements of the ARTMOS's chosen application domain — complex real-time systems such as flight control or vehicle management systems with strict time constraints and distribution requirements. Special emphasis is placed throughout the functional analysis on the multiprocessing support provided by these operating systems. The RTOSs examined are:

- Ready System's VRTX/MPV (35)
- Software Components Group's pSOS/pRISM (71, 72)
- Software Engineering Institute's Distributed Ada Real-Time Kernel (DARK) (2)
- Taurus System's Harmony (81)

Features of these real-time operating systems are compared with the AMCAD RTMOS where appropriate.

The operating system functions identified fall into eight categories: task management, communications management, semaphore management, memory management, interrupt management, time management, initialization, and file management. Each of these functions is examined in more detail in the following subsections.

3.4.1 Task Management This subsection examines the functional requirements for the management of tasks by the ARTMOS. The ARTMOS task management functionality must be defined such that it does not constrain the development of applications software or the partitioning of that software into autonomous tasks.

3.4.1.1 Tasks A *task* is a sequential unit of software that comprises a logically complete function or collection of related actions (63:34). Tasks are the smallest units of execution that can compete on their own for system resources (71:2.3). In a multitasking system, tasks can execute concurrently or sequentially with respect to each other. Because multiple tasks are running, each must have an assigned priority to determine which tasks take precedence over others for scheduling. Several tasks can operate autonomously from the same code, or tasks can each have their own unique code.

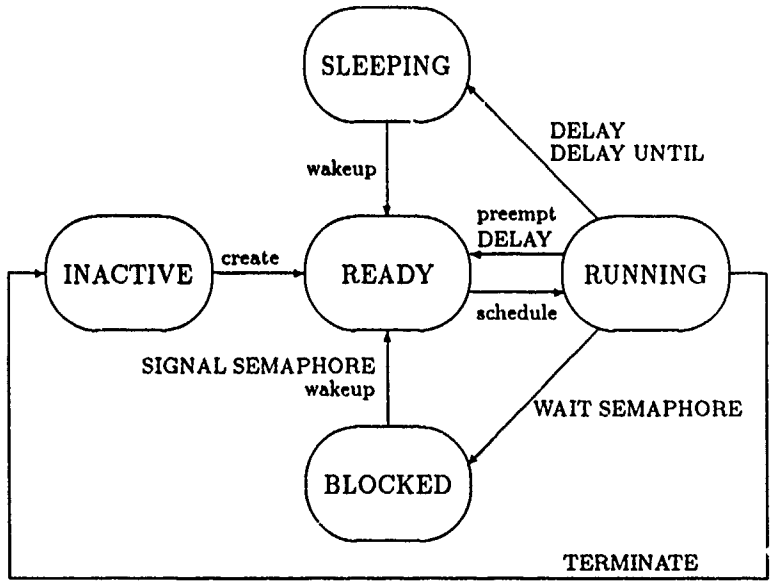


Figure 3.2. Task State Diagram

Tasks are represented in the OS by *task control blocks (TCBs).* A TCB saves the exact state or context of the task when it is not executing so that the state can be restored next time the task runs. In this fashion, the task appears to have its own processing environment.

Depending on its priority and status, a task can be either inactive, ready, running, or blocked (2, 35, 36, 64, 71) as shown in Figure 3.2. An *inactive* task is one which has not yet been created or whose execution has terminated. Inactive tasks have no TCB. *Ready* tasks are ready for execution, but must wait for an available processor. A *running* task has control of a processor and is executing its instructions. Only one task is running at a time on a given processor. Tasks *block* when they cannot complete an action they initiate. Tasks can block waiting for messages to arrive, semaphores to become available, some hardware event to occur, a specific time interval, or a specific restart time.

3.4.1.2 Scope of Task Management Functionality One element which varies drastically among real-time operating systems is the influence or scope of the task man-

agement functions. To what extent can one task affect the execution of another? With VRTX, for example, any task can suspend or resume any other task. Nothing prevents a task from wrongly shutting down another task (due to coding error, or damaged program memory, for example). VRTX extends this ability to tasks on multiple processors with *remote procedure calls*, a method of invoking operating system calls on other processors. pSOS allows the user to group related tasks together, an ability not found in VRTX. Tasks under pSOS can only affect tasks within their group. Though pSOS has a remote procedure call mechanism, task management calls are not supported remotely. DARK, on the other hand, only lets a task alter its own state.

For the real-time distributed control applications targeted by the ARTMOS, to what extent should tasks be able to affect one another's operation? From a fault tolerance point of view, it would not seem a good idea for tasks to be able to suspend or otherwise alter the execution of each other. If task A suspends or terminates task B, for example, task B may not be able to meet its deadlines. Likewise, if task A lowers the priority of task B, B may not execute in time. In systems which allow tasks full access to each other, the programmer not only has to make sure deadlock situations are avoided, he must ensure that deadlines are not missed because of some subtle task interaction. Another potential problem with allowing tasks to affect one another is lack of reusability. Tasks which suspend, resume, or alter the execution of each other are explicitly bound together, and thus must be reused as a single entity.

Both DARK and the AMCAD RTMOS provide a minimal tasking model. That is, a task can only affect its own operation. The operating system provides only those functions a task needs to control its own execution. As a result, tasks are concerned with their own behavior and not that of others, essentially "minding their own business." Such a tasking model is more straightforward and easier to program, as each task is self-contained. No other task can alter the task's priority, suspend it, terminate it, or otherwise change the execution of the task. This task model is also more predictable since a task cannot be altered by other tasks. For these reasons, the ARTMOS task management functions only affect to the calling task, as done in DARK and the AMCAD RTMOS.

3.4.1.3 Static Versus Dynamic Task Management An area of great interest in real-time systems is the allocation and scheduling of application tasks. In single processor systems, the scheduling algorithm determines some schedule for executing tasks such that all time and resource constraints are satisfied (16:151). If complete knowledge of task characteristics and start times is available, then a schedule can be *statically* computed off-line describing which tasks execute and in what order. Static scheduling has a low run-time cost, but is inflexible and cannot adapt to changing or unpredictable environments. If new tasks are added, the entire schedule must be recalculated and retested (16:151). In some cases, task characteristics such as the number of tasks, their start times, or their worst-case computation times are not known *a priori* or are not predictable. Static scheduling cannot cope with non-deterministic task behavior, so tasks must be scheduled *dynamically* at run-time. Dynamic approaches have higher run-time overhead and can be less predicatable than static scheduling, but are more flexible and adapt easily to changing environments (16:151).

Multiprocessor systems introduce the added dimension of determining which tasks execute on which processor, often called task allocation or assignment. (16, 46, 49, 59, 76). One approach is for the processors to share a common set of task queues (59:68). When a processor needs work, it requests a task from the shared ready queue. Access to the shared task queues must be mutually exclusive to prevent two processors from getting the same task at the same time. Processor utilization should be high unless contention for the shared ready queue or insufficient ready tasks keep processors waiting. Load balancing is enhanced because newly created or awakening tasks go into the shared ready queue and execute when a processor is available. This approach is a form of dynamic scheduling, and is used by VRTX and pSOS. Shared task queues, however, implies a shared memory, which may not be available in loosely coupled multiprocessor or distributed systems.

Quinn (59:68) and Cheng et al (16) suggest that a more appropriate solution would be for each processor to have its own kernel and task queues. The total system task load is partitioned such that each processor receives its own set of tasks to manage. Many static approaches exist for allocating tasks which attempt to balance the processor workloads and minimize interprocessor communication costs (16:). Dynamic approaches consist of a local

scheduling algorithm and a distributed scheduling algorithm. The local scheduler manages the local task load. If new tasks arrive or the processor's workload is too heavy, the distributed algorithm dynamically determines which processor should accept the task(s) (16:). The local scheduler can be static or dynamic regardless of the allocation scheme used. AMCAD, DARK, and Harmony dynamically schedule the tasks on each processor, but use static task allocation. As above, dynamic allocation algorithms better utilize the processors and are more adaptable to unpredictable events, but with a high communications cost at run-time. Static approaches may not utilize the resources as efficiently, but are fast and predictable. Stankovic et al (76:273) note that an optimal schedule is NP-hard and computationally intractable, particularly under run-time constraints. More practical, suboptimal algorithms based on heuristics can reduce the overhead.

For this initial prototype of the ARTMOS, what is needed for task allocation and scheduling? With respect to the research objectives, any of these approaches is sufficient. The approach of statically allocating task sets to processors and dynamically scheduling the tasks is used because it is the least complicated and is consistent with the AMCAD RTMOS. Note, however, that the AMCAD and ARTMOS task model is independent of the scheduling and allocation algorithms, so long as application time constraints are not violated. The application programmer can experiment with different scheduling approaches without compromising application task execution. Specific algorithms for scheduling and allocating tasks are discussed in the detailed design phase.

3.4.1.4 Task Management Requirements An assumption for this thesis investigation is that all code and data could be stored in one processor. That is, each processor has enough memory for the entire application. This eliminates the need for code migration and most memory management overhead. Though a processor must have the memory capacity to hold the entire application, it does not have to be powerful enough to execute the application within time constraints.

Because of the transparency requirement, system calls should be designed so that they are called the same regardless of the number of processors (one or more). Likewise, any reference to a task should not require specification of the processor on which that task

currently resides. Therefore, each task should have a unique logical identification (ID) assigned at design time that is independent of the processor on which the task is currently executing. This requirement does not preclude the operating system from assigning physical IDs to tasks local to each processor for management purposes.

Another question regards use of multiple tasks based on the same code segment or "routine." In many circumstances it would be useful to have multiple tasks performing the same function — matrix multiply, for example. Rather than duplicate the code for the number of needed tasks, it makes more sense to make reentrant code that could be used by multiple tasks concurrently. This requirement necessitates low level mechanisms to handle any "duplicate" tasks.

Task reconfiguration may be required for fault tolerance. In case a processor fails, it may be necessary to reconfigure the workload such that the other processors assume the workload of the failed processor. To implement reconfiguration, algorithms are needed for fault detection to identify the need for reconfiguration, as well as algorithms for isolating the cause of the fault and removing it. While there is a certain interest and need for such a capability, it is not needed to meet the objectives of this thesis investigation. These functions are more suitable for follow-on efforts.

3.4.1.5 Task Management Functions The following is an in-depth examination of the potential task management functions. Table 3.1 lists common task management functions for the real-time operating systems examined.

All three operating systems (other than the AMCAD RTMOS) implement preemptive scheduling, with time-slicing an option for tasks of the same priority. VRTX assigns each task a unique identification number (ID) which is used to refer to a task. pSOS and DARK assign each task a unique physical ID when the task is created. Tasks can be referenced by the physical ID or the logical task "name" assigned by the user at design time.

TERMINATE — This function removes one or more tasks from the set of active tasks. The deleted tasks becomes dormant and the TCB is available for reuse. For this task to execute again, another task or the OS must *create* it again. Since under the

System Call	VRTX	pSOS	DARK	AMCAD
DELETE/KILL	X	X	X	X
CREATE/ACTIVATE	X	X	X	X
SUSPEND/RESUME	X	X		
GET/SET PRIORITY	X	X	X	
GET/SET PREEMPTION	X	X	X	
GET/SET TIME SLICE	X	X	X	
DELAY	X	X	X	X
WHO AM I			X	
NAME OF			X	
GET STATUS	X	X	X	

Table 3.1. Task Management Calls

ARTMOS tasking model no task can directly affect another, a task can only terminate itself.

CREATE — This function creates a task by getting an available TCB, loading it with the task's initialization data, and scheduling the task. One critical issue is whether tasks are created statically at compile time or during initialization, or dynamically at run-time. In dynamic creation, tasks create "children" tasks as needed and terminate them when they complete, or the children terminate themselves. The dynamic approach increases the run-time cost of creating and terminating subtasks, but avoids the constant overhead of managing many pre-created tasks. One hybrid approach, used in pSOS, allows creation (allocation and set up of TCBs) of all tasks at initialization and *activation* (load into task queues and schedule) of tasks at run-time as needed, reducing the run-time cost. Dynamic task creation binds the tasks more tightly to one another, reducing the potential reuse of code within an application.

Dynamic task creation is even more complicated for parallel systems. For example, where is the new task assigned? Is it assigned to the processor with the creating task or the processor with the lightest workload? A dynamic scheduling algorithm, as mentioned above, would be needed to determine the best processor to receive the new task. DARK avoids this problem by creating the new task on same the processor as the calling task. But what if the local processor is already overloaded? Another potential problem with a local create is an inability to take advantage of application parallelism. If a thread of

computation is formed by each task spawning off subtasks, no parallelism will ever be exploited: all components of a thread would end up on the same processor.

AMCAD creates all tasks at initialization. Tasks sleep on the timer queue until they are ready to execute. The only entity that can create a task is the OS at (re)configuration or for a special event, such as an error condition. The ARTMOS follows this static task creation model because it supports the earlier decisions to implement the AMCAD client/server task model and statically assign tasks to processors. As before, this decision does not preclude the addition of dynamic task creation at a later date; it is simply the simpler option. Therefore, the OS can create tasks, but the application cannot.

SUSPEND/RESUME — VRTX and pSOS allow any task to suspend operation of any other task (or subset of tasks) and resume them later. Not only do these operations violate the ARTMOS tasking model, they also have potential for deadlock or missed deadlines. Suppose task A has suspended tasks B and C. Task D then suspends task A. Tasks B and C do not resume. This situation should be avoided, particularly in systems where fault tolerance is important. In addition, how much more difficult is it to verify the real-time operation of a system when tasks can be arbitrarily suspend one another? SUSPEND and RESUME are not implemented.

GET/SET PRIORITY — These primitives allow a task to check or modify the priority of a task. In VRTX and pSOS, any task can change the priority of any other; DARK only lets a task alter its own priority. As stated above, ARTMOS calls affect only the calling task. When would a task need its priority? If a task is the only entity authorized to change its priority, it should already know its priority. It is possible, however, that the OS may want to alter priorities as task deadlines become more urgent or to prevent priority inversion. The ARTMOS thus needs GET PRIORITY and SET PRIORITY primitives.

GET/SET PREEMPTION MODE — These functions give tasks some control over how they are scheduled. SET PREEMPTION MODE could be used to prevent a task from being preempted during critical sections, and reenable preemption after the critical section. Entire tasks could be non-preemptable for their "lifetime." As before, these functions work for the calling task only. In this case, GET PREEMPT MODE is not really needed, since

a task's preemption mode cannot be altered by another task and the OS has no reason to prevent a task from being preempted. Therefore, a SET PREEMPTION MODE primitive is required.

ENABLE/DISABLE TIMESLICE, ALTER SLICE — Time slicing supports round-robin scheduling of equal priority tasks by giving each a fixed amount of processor time. When the time expires, the running task is switched out and replaced with another. Time slicing is optional in VRTX and pSOS and not supported by DARK or AMCAD. VRTX allows user tasks to enable or disable time slicing, but has no means of setting the slice time. Tasks under pSOS can give up the processor (and cause a round-robin) by pausing for zero time. This releases the processor so that any waiting tasks of the same priority can execute. If no equal priority tasks waiting, the calling task gets the processor back.

To what extent can a task inhibit or enable time slicing? If a task needs to prevent itself from being switched out, it could set its mode to non-preempt, preventing itself from being switched out until ready. More globally, a task could disable or enable time slicing for the entire processor. In general, time slicing gives equal priority tasks a "fair" chance to run. If application tasks are written as single functions and tasks don't "hog" the processor, time slicing should never be needed. Tasks that do perform long computations could release the processor at intervals (as in pSOS). For these reasons, time slicing is not implemented.

WHO AM I — This primitive allows a task to ask for its physical identification assigned by the OS. Under the AMCAD and ARTMOS tasking models, tasks refer to one another exclusively with logical names assigned at design time. In addition, a task only needs the addresses of its input and output variables to do its job. A WHO AM I primitive is not required. Likewise, a NAME OF function to get the ID of another task is not required since tasks cannot address another task directly.

DELAY/DELAY UNTIL — Two important functions allow tasks to wait for an interval of time or until a specific time. This is particularly important as much of flight/vehicle control computation is periodic — updating flap position, updating stick position, and so on. Certain computations need to be performed at regular intervals to remain accurate.

These two functions provide a means of implementing periodic activity. Note that a task can release the processor by waiting for zero time. The ARTMOS implements DELAY and DELAY UNTIL primitives.

3.4.2 Communication Management Many RTOSs, such as pSOS and VRTX, provide different communication models for tasks on the same and different processors. As such, the application programmer must consider the physical location of tasks (which processor) when coding application tasks to know which communication routines to use. One reason for this asymmetry is that pSOS and VRTX are single processor RTOSs. Both provide an optional package for shared memory multiprocessor systems. The OSs were not designed for consistent, transparent parallel processing support, and as a result they have different sets of system calls for local and global operations. For a clearer picture of some common communications schemes, the following examines communications in VRTX, pSOS, DARK, Harmony, and the AMCAD RTMOS.

Single processor VRTX provides *mailboxes* and *message queues* for intertask communications. A mailbox is a user-defined variable through which tasks can pass fixed-length messages or pointers to longer messages (35:). Mailboxes hold one message at a time. Tasks can *post* messages to a mailbox and *pend* or *accept* to receive a message. If a task tries to post a message to a full mailbox, the post is igored and the task is returned an error message. Pend suspends the calling task if the mailbox is empty, while accept returns an error code but does not block the task. If multiple tasks are pending on a mailbox, an arriving message goes to the highest priority waiting task. Mailboxes can be used as semaphores by forcing tasks desiring a resource to pend on a mailbox. A task releases the resource by posting a message to the mailbox (35:2-20). A message queue is a fixed-length buffer created and managed by VRTX via system calls initiated by the application. Unlike mailboxes, message queues can store many messages (up to some limit) until they are requested. Tasks can post, pend, and accept messages from queues as they would from mailboxes. Messages are given to pending tasks based on the priority of the tasks. A task creates a message queue by issuing a *qcreate* call, and can inquire on the status and length of the queue.

VRTX multiprocessor communications (with the MPV package) is based on *channels* between processor nodes. MPV supports any number of channels between nodes, so applications can select the appropriate communication model, whether task-to-task, node-to-node, or some hybrid. MPV calls allow tasks to create channels, send messages to channels, receive or accept messages from channels, and check channel status. MPV also provides *remote procedure calls (RPC)*, constructs that allow tasks to invoke system calls on other processors through the channels. All VRTX system calls can be invoked remotely with this facility. RPCs do cause some overhead, however, as each node needs server tasks to perform the call and send any results to the remote node.

pSOS also has two single processor communications and synchronization mechanisms: *message exchanges* and *events*. Exchanges are essentially the same as VRTX message queues, though the pSOS system calls give the user more flexible access to the exchanges. Messages can be sent to the front or rear of the exchange queue, or broadcast to all blocked tasks (waking up multiple tasks with one message). When a task requests a message, it can elect to wait, wait with timeout, or return unconditionally if no message is pending. Blocked tasks can be ordered by arrival or priority. Unlike messages, pSOS *events* carry no information and are used primarily for synchronization. Event flags signal whether or not specified events have occurred. Tasks wait on events much as they would for semaphores. What characterizes events, however, is that tasks can wait on one event, all of combinations of events (AND), or any of a combination of events (OR). While messages are sent to exchanges without knowing the destination task, events are directed at specific tasks. A task can *signal* events to other tasks, *wait* for events, or *get* events to check or reset flags.

A separate pSOS package, pRISM, can be added for shared memory multiprocessor support. Interprocessor communications is based on *datagrams* of variable length. Each processor node has a single queue in shared memory for incoming datagrams. To send a message, tasks request a buffer from pRISM, load the data, and use pRISM calls to place the message at the front or rear of the destination node's datagram queue. On the sending node, pRISM interrupts the receiving node to indicate a datagram is present. A datagram interrupt handler on the receiving node dequeues the datagram. If interprocessor

interrupts are not available, each node must poll for inbound datagrams. pRISM leaves it to the application to determine what happens to datagrams once they are received (such as which task gets the datagram). Like VRTX/MPV, pRISM supports remote system calls, which are implemented with special datagrams. pRISM, however, only supports ten remote pSOS calls. These calls can initiate actions on other processors, but they cannot prevent actions. For example, a task on one node can resume a remote task, but it cannot suspend a remote task. Remote pSOS message calls may be more useful than datagrams, as messages can be directed to specific tasks, not just the node (71:2.13).

pSOS and VRTX are single processor RTOSs, with extensions for multiple processors, resulting in different primitives for communicating between tasks on the same and different processors. DARK, on the other hand, is designed specifically for distributed systems and emphasizes consistent task communications, regardless of the number of processors. Rather than provide several communications mechanisms, DARK provides a single, task-to-task communication model based on message passing. If task A on processor node 1 has a message for task B on node 2, it gives the Kernel on node 1 the message and a logical destination (task B). The Kernel builds a *datagram* and sends it to node 2, where it knows task B resides. The node 2 Kernel rebuilds the message from the datagram and queues it until task B requests the next message. If the message requires a response, the node 2 Kernel formats an acknowledgement datagram and sends it back to node 1 when task B receives the message. Communications primitives are RECEIVE MESSAGE, SEND MESSAGE, and SEND MESSAGE AND WAIT for acknowledgment. RECEIVE and SEND MESSAGE AND WAIT are blocking primitives, causing the calling task to block until the action is complete or a timeout expires. Programmers can choose synchronous or asynchronous communications, as their application dictates. DARK transparently optimizes communications between tasks on the same processor.

Taurus Computer Products' Harmony, a multitasking, multiprocessor OS for real-time embedded control, uses synchronous message passing for intertask communications and synchronization. A sending task blocks until its message has been received and acknowledged. Receiving tasks only block if no messages are available. Harmony uses SEND, RECEIVE, and REPLY primitives to implement the message passing. Synchronous mes-

sage passing, according to Taurus, removes the need for message queues (and the problem of buffer overrun), and promotes the design of robust, modular software. Software debug is easier because only one message from a sending task is active at one time.

Like DARK and Harmony, the AMCAD RTMOS offers a single, consistent communication model independent of the number of processors or physical location of tasks. The AMCAD model is similar to the VRTX mailbox scheme, but with several variations. In the model, every *global* data variable — data items passed between tasks — has a unique *mailbox* in the Virtual Common Memory consisting of producer and consumer flags, the data size, and the data itself. A producer task "sends" a data item to its mailbox; likewise, a consumer "receives" the data item from its mailbox. A producer/consumer algorithm protects the integrity of the data exchange, preventing data from being consumed before it has been produced, and so on. The producer/consumer algorithm is built into the SEND and RECEIVE primitives to make it transparent to the programmer. Send is a non-blocking primitive; if the previous data hasn't been consumed, the task overwrites the old value and signals the error to the OS. If a task attempts to receive from an "empty" data variable — one that hasn't been produced since last consumed — the task suspends until the data is produced or a timeout occurs. If the data is still unavailable after a timeout, the task consumes the previous value and reports the error to the OS. Communications is thus asynchronous, as only the receiving task waits. Mutual exclusion is not needed on a variable's producer/consumer flags since each variable has only one producer and one consumer task.

Which communication model is most appropriate for the complex hard real-time applications targeted by the ARTMOS? First of all, the ARTMOS should follow the AMCAD programming model as closely as possible to meet the objectives of this thesis investigation. By designating unique mailbox structures for each global data variable, the OS simplifies the design of application software. Tasks can be designed without knowledge of any other tasks; a task only needs to know which global data variables it produces and consumes. The producer/consumer algorithm indicates when data is available or consumed, an important consideration in a potentially distributed system without interprocessor interrupts. From examination of DARK and Harmony, it appears that there may be a need for a block-

ing SEND primitive in addition to the asynchronous SEND and the blocking RECEIVE. This primitive is implemented, but with the recommendation that the programmer use the non-blocking SEND whenever possible for determinism. The ARTMOS provides blocking SEND and RECEIVE primitives, along with the non-blocking SEND. All blocking calls are bounded. Note that a non-blocking RECEIVE can be done by passing a wait duration of zero.

3.4.3 Semaphore Management Semaphores are perhaps the most common means of synchronization as they are easy to implement and can be used to build higher level constructs. A semaphore is a simple shared flag with two associated operations, *wait* and *signal*, that can be used for mutual exclusion or resource sharing (59:67). In its simplest form, the semaphore flag is either *free* or *claimed*. A task attempts to claim the semaphore with the wait operation. If the flag is *free*, the semaphore changes to the *claimed* state and the invoking task continues; otherwise, the invoking task blocks. The task with the semaphore uses the resource or executes the critical section, then releases the semaphore with the signal operation. If any tasks are blocked waiting for the semaphore, one is granted the semaphore and allowed to proceed. Otherwise, the semaphore is freed. The wait and signal operations must be atomic so that only one task at a time can claim the semaphore (4:50). This is known as a *binary* semaphore. Other types of semaphores, such as counting semaphores (4:50), allow sharing of multiple resources of the same type.

Neither VRTX nor pSOS explicitly support semaphores, but both can implement semaphores with the functions they do provide. DARK and the AMCAD RTMOS provide semaphore support, but the semaphores are visible only on the local processor. That is, semaphores can only be used by tasks on the same processor. Table 3.2 summarizes semaphore support in the operating systems. Global semaphores, for mutual exclusion of tasks on different processors, are straightforward in systems with shared memory since the semaphore can be shared by the processors. In a potentially distributed environment, where a shared memory space cannot be assumed, mutual exclusion is extremely difficult. For this reason, and because tasks may need the ability to contend for local resources, the ARTMOS implements local semaphore WAIT and SIGNAL routines. To prevent tasks

System Call	VRTX	pSOS	DARK	AMCAD
WAIT			X	X
SIGNAL			X	X

Table 3.2. Semaphore Management Calls

from blocking indefinitely, the ARTMOS allows tasks to specify a timeout value with the WAIT operation to bound the wait time.

A number of algorithms exist for mutual exclusion in a distributed environment (48:205–225). All require at least $N/2$ transmissions, for N processors, and can be very time consuming. Depending upon the communications overhead of the target system, the distributed mutual exclusion algorithm chosen, and the hard time constraints of the real-time application, the time to guarantee distributed mutual exclusion may result in missed deadlines. It is possible, however, that certain physical devices in the flight control system might require controlled access by tasks on different processors, or in different subsystems. In this case, distributed mutual exclusion would be necessary. This capability is beyond the scope of this thesis investigation. Later addition of this capability would not change the way the system calls are made, though the calls themselves would be implemented quite differently.

3.4.4 Memory Management The memory management portion of the OS tracks which parts of memory are in use, allocates memory to tasks which need it, and deallocates memory when the tasks complete (80:291). The sharing of memory among tasks, however, must not cause erratic task execution times which could result in missed time constraints. Since virtual memory techniques such as paging can create large and unpredictable delays, they are not suitable for hard real-time systems (76:378). For this reason, the ARTMOS does not consider virtual memory schemes. Table 3.3 shows several memory management calls.

Two major memory management schemes are possible, both of which divide available memory into partitions which are allocated to tasks as needed. The difference between the two schemes is whether the partitions are of fixed or variable size (56:158-164). In fixed

System Call	VRTX	pSOS	DARK	Harmony	AMCAD
CREATE PARTITION	X	X			
DELETE PARTITION	X	X			
GET FIXED BLOCK	X	X			
FREE FIXED BLOCK	X	X			
GET DYNAMIC SEGMENT		X		X	
FREE DYNAMIC SEGMENT		X		X	

Table 3.3. Memory Management Calls

(static) partitioning, the partitions may be different sizes or all the same, but cannot change size during system operation. The problem with fixed partitions is determining the best partition sizes to minimize fragmentation of the available memory, often a precious resource in real-time systems. With variable (dynamic) partitioning, tasks are allocated the exact amount of memory they request (56:164-171). Memory utilization is generally better with variable partitioning, but memory may need to be compacted periodically to collect unused portions.

VRTX and pSOS implement a hybrid of these two approaches, allowing applications to create dynamically sized partitions of fixed size blocks. Once the partitions have been created, time to allocate a block is short and deterministic, yet memory fragmentation is reduced by using dynamic partitions. pSOS and Harmony allow tasks to request variable size memory blocks, a capability VRTX does not support because this approach is slower and less predictable. Unlike pSOS, Harmony does not provide any static memory allocation functions. The DARK has no memory management mechanisms at all: variables and data structures are statically allocated at initialization and so are available quickly and when needed.

For real-time systems, when is task memory needed? A task's memory requirements include the task control block (TCB) space, stack space, and a work area. While the TCB and stack space remain constant, a task's need for work space varies over the course of its execution, and different tasks have different requirements. VRTX and pSOS allocate TCB and stack space to tasks as they are created, with work space being requested by the tasks as needed. If a request for memory cannot be met under VRTX and pSOS, the implication

is that the requesting task(s) waits until memory becomes available. In real-time, this unpredictable delay may result in missed deadlines.

If each processor is assumed to have enough available memory for all task control blocks, stacks, and work space needed by the application, then all variables and data structures can be allocated at initialization and memory management is not necessary. This assumption may not always be reasonable. A processor does not need to provide enough memory for the entire application's work space, only enough for the maximum workload the processor will have. For example, in a fault tolerant system, if the total system workload is partitioned onto N processors, each processor needs enough memory for its share of the workload. However, as processors fail or are damaged, the workload is redistributed onto fewer processors until not enough healthy processors remain. At that point, the application reduces to a degraded mode capable of running on the available processors. Each processor must have enough physical memory to meet the demands of its maximum workload. In addition, Stankovic and Ramamritham (76:378) note that many complex real-time systems (such as flight control or spacecraft avionics) consist of disjoint phases (pre-flight, take-off, cruising, descent, etc.). The amount of memory needed must be enough for the largest phase, not the entire system.

Memory management may be needed in the ARTMOS, so it is provided. Because of the lower predictability and higher overhead of dynamic memory management, memory is managed as fixed size partitions. Note that there could be several different size groups. One might be TCB-sized, another stack-sized, and so on. The decision to implement static memory management should not preclude later implementing dynamic algorithms, if the designer is willing to pay the penalty. For this implementation, the ARTMOS provides a fixed block memory management scheme, with ALLOCATE MEMORY and DEALLOCATE MEMORY system calls.

3.4.5 Interrupt Management Because real-time control systems manage the operation of environments external to the computer system, some means is needed for the environment to notify the system when certain events occur. One commonly used mechanism is the interrupt. Operating system support for interrupts can be difficult because

of the variety of possible interrupts, particularly when different applications may need the same interrupt handled quite differently. A more manageable solution is for the operating system to provide a mechanism for binding application-supplied interrupt handlers.

Such a mechanism requires two different activities. First, the application needs some way to point the OS to specific interrupt or exception handlers. Secondly, the OS must be notified when the handlers complete. If the handler could have affected the order of task execution, the OS needs to reschedule before returning control to the application tasks. Otherwise, the interrupted task resumes executing where it left off, even if the handler readied a higher priority task. For example, an interrupt from a hardware timer might increment the processor's clock variable, then "wake up" any tasks waiting until the new time. In this case, a task with a higher priority than the interrupted task may enter the ready state. If the handler returns control to the interrupted task without notifying the OS, the higher priority task is blocked by the lower priority task.

For these reasons, the ARTMOS supports application interrupts by providing two system calls, BIND INTERRUPT HANDLER and RETURN FROM INTERRUPT. The BIND INTERRUPT HANDLER call maps the handler to the specified interrupt and has the advantage of making the definition of interrupt handlers explicitly visible in the application code. RETURN FROM INTERRUPT passes control to the OS after a handler completes so the OS can reschedule if necessary.

None of the real-time operating systems support interprocessor interrupts. The operating systems only provide handling capability for local processor interrupts. That is, an interrupt that occurs on a processor is handled by that processor, and one processor cannot interrupt the operation of another. Because of the possibility for conflicts between interrupt handlers, there may be a need for an ability to enable or disable specific application interrupts. DARK has two system calls which provide this capability. The ARTMOS also supports this ability.

Four interrupt functions are therefore required. They are: ENABLE INTERRUPT, DISABLE INTERRUPT, BIND INTERRUPT HANDLER, and RETURN FROM INTERRUPT.

3.4.6 Time Management A critical aspect of real-time systems is *time*. The system must perform certain actions based on time constraints derived from the "real" environment outside the system — in the case of flight control, the aircraft. If the system clock is incorrect, these deadlines may be missed. Much of the computation in the system also relies on time — whether in a time-based function or priorities which increase as deadlines approach.

Given the importance of time in real-time systems, to what extent can or should tasks affect the system clock? As shown in Table 3.4, VRTX, pSOS, and DARK all allow application tasks to set or adjust the local processor clock value. With this ability, a task can alter another's operation by changing the clock. For example, if task A is waiting for time 32, task B could start task A prematurely by setting the clock to 32. The result is wasted processor time and potentially missed deadlines. A SET TIME function could be extremely dangerous without some control over which tasks can use it and in what circumstances.

System Call	VRTX	pSOS	DARK	AMCAD
SYNCHRONIZE			X	
GET TIME	X	X	X	X
SET TIME	X	X	X	
POST TIME INCREMENT	X	X		

Table 3.4. Time Management Calls

The POST TIME INCREMENT system call allows the user to supply a clock interrupt handler. The handler would be able to inform the OS of the time increment. The motivation for this function in VRTX and pSOS is that these operating systems assume only the processor and memory. Other devices, such as a real-time clock, must be initialized and managed by the application. The AMCAD RTMOS and DARK assume that since a clock/timer is a critical part of the real-time system, it will be present and managed by the operating system. The application software in this situation does not need a POST TIME INCREMENT call.

Multiprocessor systems can complicate time management, depending on whether the processors share a common clock or have independent clocks. Fault tolerant real-

time multiprocessor systems, such as flight or vehicle control systems, may require that each processor have an independent clock to avoid a single failure point. For systems with distributed clocks, the issue of synchronization is crucial. The SYNCHRONIZE call in DARK forces all processors to set their clock value to that of the calling processor. This is a potentially dangerous situation, for if the calling processor is overly fast or slow, all processors become incorrect. In a flight critical system, it is not important that the processor's clocks are consistent with one another, but that they are within some delta value of the "true" system time. For these reasons, application tasks should not be able to set or alter their local clock value. The OS is responsible for ensuring that the local clocks are synchronized, not the application.

The operating system, therefore, has two responsibilities with respect to time management. First, the ARTMOS provides a call to read the local clock, GET TIME, as an application might need the time for a computation or as an alarm offset for delaying. Second, the operating system must keep the local clocks synchronized within T clock ticks.

3.4.7 Initialization There may be a need for some application-specific initialization, such as the the binding of interrupt handlers or definition of semaphores, which must take place before the OS begins the multitasking of application tasks. Recognizing the need for both operating system and application initialization, VRTX, pSOS, and DARK all provide initialization protocols.

With the single processor VRTX and pSOS approaches, system execution begins with application initialization code. When the application code is ready, it triggers the OS to initialize and begin multitasking. The difference between the two approaches is VRTX implements OS initialization and startup as two separate system calls, while pSOS combines the two actions in one call. Since the VRTX method separates OS initialization and startup, the user can use VRTX calls during application initialization by initializing the OS first.

Multiple processor systems require initialization of both local (to each processor) and global (system-wide) resources. VRTX, pSOS, and DARK all provide some means of designating one processor as master and the others as slaves for synchronous system

initialization. The master initializes any shared resources (shared memory for VRTX and pSOS, the network for DARK) and directs the slaves to perform local initialization. When initialization is complete, the master triggers all processors to pass control to the application code. The application, after completing its startup code, signals the OS to begin multitasking. Master/slave initialization functions might include calls to allow the master to download application code, or perform system-wide self-test.

The approach in ARTMOS is somewhat different from those described above. Upon reset, each processor node initializes local resources, such as timers, memory, and data structures. When this initialization completes, control is passed to the application initialization code, which performs application-specific initialization. When the application is ready to begin multitasking, an INITIALIZE MASTER call is issued on the master processor. The other processors issue INITIALIZE SLAVE calls. The master processor initializes any global resources, then triggers the slaves to select a set of application tasks and load them. When the tasks are ready, multitasking begins.

In ARTMOS, then, functions are needed to trigger master and slave initialization. It is assumed that each processor is loaded with all application code and data at system initialization. The system calls provided are INITIALIZE MASTER and INITIALIZE SLAVE.

3.4.8 File Management Many real-time operating systems provide some file management capability. VRTX and pSOS, for example, have optional packages for establishing and managing a file system. In both cases, the file system is local to each processor; in a multiple processor system, any processor with the file management package could have its own file system. Neither provides true distributed file system capability, though the VRTX/MPV remote procedure call facility allows tasks to access files on remote processors. DARK, a real-time kernel and not a full blown operating system, does not support file management.

To what extent is a file system needed in a real-time, embedded control system? Most real-time systems that need file systems are interactive or based on standard workstations for process control or real-time data acquisition. In such systems, the application code

may need modification on-line or continuous data storage, thus requiring a file system. Embedded systems typically do not require file systems. A fully tested and operational ARTMOS would not require a file system. During development and testing, however, the ability to open and write to files could be valuable for performance monitoring and debugging. Each processor could store performance data which would be periodically stored to disk (freeing memory for more data). Though a useful feature, a file system is not essential to meet the project objectives.

3.4.9 Summary of Functional Requirements This subsection summarizes the operating system primitive functions that the ARTMOS is required to provide. The primitives are listed in Table 3.5.

Task Management	*Communication Management*
TERMINATE GET PRIORITY SET PRIORITY SET PREEMPTION MODE DELAY/DELAY UNTIL	SEND SEND AND WAIT RECEIVE
Semaphore Management	*Memory management*
WAIT SIGNAL	GET MEMORY FREE MEMORY
Interrupt Management	*Time Management*
ENABLE INTERRUPT DISABLE INTERRUPT BIND INTERRUPT HANDLER RETURN FROM INTERRUPT	GET TIME
Initialization	
INITALIZE MASTER INITIALIZE SLAVE START ARTMOS	

Table 3.5. ARTMOS System Calls

3.5 Performance Requirements

As discussed in Chapter 2, one important characteristic of real-time embedded systems is the demanding performance requirements. These systems must be able to respond

to external events within a fixed amount of time, or provide output at strictly defined intervals. The urgency of these time constraints is derived from the particular application environment, be it flight control, command and control, or factory automation. In a flight critical environment, the penalties for missing deadlines can be fatal. If radar data is not read at the proper times, for example, the data is simply overwritten by new values, with perhaps no performance degradation. The control laws for statically unstable aircraft such as the X-29, on the other hand, must be updated on time or the aircraft crashes. Because of the high performance and predictability requirements of such real-time systems, ARTMOS operations must be fast, efficient, and predictable, and should support the types of performance requirements typical of these applications.

The specification and design of the ARTMOS in this thesis project is not intended to be a production system; rather, it provides a laboratory research capability. The initial development targets a multiprocessor system available in the AFIT environment, not an actual embedded system. While use of laboratory computers enhances the research value of the ARTMOS by providing software development and debugging tools, they are not able to meet demanding real-time performance requirements. As such, it is not really possible or feasible to place specific timing requirements on this first implementation.

Rather, this section identifis performance metrics of importance for the operation of the ARTMOS. The OS development attempts to design in performance "hooks" to allow measurement of system performance for later analysis. These performance metrics may also be helpful in evaluating different OS algorithms or optimizations for a particular architecture. Possible performance metrics of interest include:

- task context switch — the time needed to switch the processor from attending one task to attending another.
- message format time — the time for the OS to format a user message for transmission.
- message receive time — the time to provide a task with a received message.
- semaphore claim/release time
- task queue handling time
- return from interrupt handler overhead

Note that some measures, such as int apt latency or message transmission time between processing nodes, are hardware dependent and unaffected by the OS software. The metrics mentioned above described potential OS overheads which must be examined.

3.6 Software Engineering Requirements

Software engineering methodologies have evolved in recent years as a means of managing the growing complexity of software systems. The purpose of software engineering is to provide a consistent, disciplined approach to the software development process to produce software that is modifiable, efficient, reliable, and understandable (9:25). For these reasons, application of good software engineering principles to the specification and design of the ARTMOS is considered essential.

One of the objectives of this thesis is that the ARTMOS provide a basic operating system framework building upon the AMCAD application program interface. This framework forms the basis for transportation of the ARTMOS to different architectures and for further experimentation of distributed operating systems. The application of good software engineering practices can help meet this objective. This section describes the software engineering principles which must be applied in this thesis project: abstraction, information hiding, and modularity; documentation; and use of a high-order language.

3.6.1 Abstraction, Information Hiding, Modularity Abstraction is the process of decomposing a problem into simpler components. This process continues until the individual components are simple enough to be reliably implemented. The essential details and properties at each level of decomposition are emphasized (abstraction) while all non-essential details are omitted (information hiding) (9:28). The programmer has access to only that information necessary to understand the level of abstraction. Abstraction enhances the reliability, maintainability, and understandability of a system by reducing the amount of information needed by the programmer at each level and by preventing operations that violate the logical view of that level (9:29).

Modularity results from the implementation of abstraction. The problem is decomposed into different functional *modules* which can be separately coded, compiled and tested;

are limited in size and function; and which may be reusable in other applications (32:434). Two measures of the effectiveness of modularity are *cohesion* and *coupling* (9, 32, 98). Cohesion describes how tightly bound or related the elements of a module are to one another, while coupling is a measure of how strongly modules are connected. Loosely coupled modules require less information to understand one another, allowing the modules to be coded, tested, and maintained relatively independently. If the elements of a module are closely related, the module is highly cohesive and more understandable, reusable, and maintainable. Effective modularity can therefore increase the reliability, testability, and understandability of a program, as well as limiting the effects of changes to a small set of modules (9:30).

Abstraction, information hiding, and modularity are important not just as sound software engineering principles, but they also are essential to meeting the objectives of this thesis investigation. The ARTMOS specification and design should provide a means of implementing the AMCAD programming model onto different parallel architectures. Use of these three principles in all phases of the ARTMOS development should enhance the portability of the design and simplify the process of "optimizing" the OS to new target architectures. An effective modular design can also provide a framework for experimentation with different OS algorithms.

3.6.2 Documentation An aspect of software development that is often overlooked is the need for accurate documentation throughout the development life-cycle. Too often, documentation is an afterthought, comments added to software to explain the flow of the code or a user's manual describing how to use the program. Documentation of this sort may be adequate, if the software is perfectly designed and implemented, and the requirements remain fixed for the life of the program. These assumptions rarely apply.

When software fails to operate correctly in a given situation, it must be modified to remove the deficiency. If requirements change and the program no longer meets performance or functional needs, some or all of the software must be redesigned and rewritten. If documentation is limited or non-existent, maintenance personnel must figure out how the software works and why before they can even *attempt* to modify it.

Good documentation spans all stages of the software development life-cycle: requirements definition, design, implementation, and testing. It provides a complete and accurate description of the decisions made and lessons learned at each step of the way so that the final product is easily understood, used, and modified. Such documentation provides insight into the selection of the initial requirements, why they were important, and how they may have evolved over the course of the development effort. It describes how the requirements influenced the specification and design of the system, along with problems encountered as the detailed design evolves and how they were resolved. Finally, good documentation discusses how the software is tested and presents results and conclusions.

All phases of the software development life-cycle must be properly documented to produce software that is truly maintainable. This thesis document serves as the documentation for the development of the ARTMOS. Because one objective of this project is that the ARTMOS be used for continued research in such areas as RTOS application program interfaces, operating system design, and multiprocessor software, complete documentation is essential. The AFIT/ENG Software Documentation Standard is used to guide the development of appropriate documentation for this research effort.

3.6.3 High Order Language The conventional approach to developing real-time operating systems has been to write the code in the "native" assembly language of the target processor. Use of assembly language supports the low-level operations needed to interface with hardware devices and can be hand-optimized to minimize code size and execution time. High-order languages (HOL) such as FORTRAN and Pascal were not used because they could not easily interface with the hardware and compiler-generated code tended to be bulkier and less efficient than hand-written assembly code.

In recent years, new languages such as Ada and C have addressed some of these deficiencies. Both provide low-level mechanisms to interface with I/O and peripheral hardware devices. Compiler technology has continued to improve and mature, so that today, compiler-generated assembly code is almost as efficient as its hand-coded counterpart. In addition, many of the newer microprocessors have instruction sets designed to work with high-order languages (60:66).

Advanced RISC (Reduced-Instruction-Set-Computer) processors, which are increasingly being used in real-time embedded applications (94:88), provide extra incentive for the use of a HOL. RISC processors are intended to be accessed through powerful optimizing compilers to ensure efficient use of the processor pipeline. As a result, "It is extremely difficult (complex, time consuming, and expensive) for unassisted humans to generate and/or optimize large quantities of RISC assembly code" (60:66).

High-order languages have many advantages over assembly language for real-time systems. Perhaps the most significant advantage is portability. Software written in assembly language for the Motorola 68030 microprocessor, for example, must be rewritten in its entirety for use on the Intel 80386. If the software is written in a HOL, it can be ported to another processor simply by recompiling it, assuming that the code has no compiler or hardware dependencies. This feature has significant effects on RTOS development. Since most RTOS are written in the assembly language of their target processor, they cannot be easily ported to a different processor (66:36). This portability can also enhance the reliability of software which is ported to other systems. Because software is recompiled rather than rewritten, the consistency and reliability of the original, proven code is maintained.

The JMI C Executive real-time kernel, for example, is written almost exclusively in machine-independent C, with the other 5% consisting of time-critical routines written in assembly and device-dependent C. Porting to a new processor requires rewriting only a few hundred lines of code, a process which takes weeks rather than the years it would take to rewrite the entire kernel (60:65). Only the rewritten code must be debugged and validated.

The control structures and English-like syntax of high-level languages make them much more readable than assembly language, which is often cryptic and can differ significantly between processors. Straightforward control-flow constructs support structured programming techniques, while "assembly languages provide no inherent structure" (9:35). The programmer does not need to worry about implementing these constructs himself, as the compiler forms the appropriate assembly code. The increased readability and more structured language features tend to make software easier to implement and test. In addition, the software is easier to maintain because it is easier to follow. It should be noted that these benefits do not appear automatically from use of a HOL. The programmer can

still write unclear or poorly constructed software, if proper software engineering techniques are ignored

3.7 Chapter Summary

This chapter presents an in-depth analysis of the ARTMOS requirements. The analysis begins with a discussion of the AMCAD Application Interface and how it affects the specification and design of the ARTMOS. Next, a model of the hardware architecture assumed by ARTMOS is developed. Several existing real-time operating systems are then analyzed to identify functional requirements — *what* the ARTMOS must be able to do and the manner in which these functions are performed. The performance requirements of the ARTMOS are considered, followed by a review of software engineering principles and how they are to be applied in this research.

The next chapter uses the system requirements developed above to produce a functional requirements specification, which defines the actions the software must accomplish to meet the system requirements.

IV. Requirements Specification

This chapter corresponds to the requirements specification phase of the Software Development Life-cycle shown in Figure 4.1. Section 3.4 defined specific requirements for the AFIT Real-Time Multiprocessor Operating System (ARTMOS) intended to describe what the ARTMOS must do without considering or constraining how the ARTMOS meets these requirements. The requirements in Section 3.4 are in the form of written descriptions of what the ARTMOS must do when completed and guidelines for its development. These written requirements must now be transformed into more formal requirements specifications defining actual activities the software must perform, data structures to be processed, and the structure necessary to control ARTMOS operation (23:173).

This chapter begins by examining several requirements specification methodologies to determine which is "best" suited for the ARTMOS specification. Next, the chosen methodology is described briefly to familiarize the reader. The third section presents key elements of the final specification and discusses problems encountered during the specification process. Finally, the specification is summarized and the design process is previewed.

4.1 Selection of Specification Method

Chapter 3 presented and analyzed the requirements for the ARTMOS. In this section, an appropriate methodology is selected for formally specifying these requirements. First, the section defines what is needed from a specification methodology. Once these requirements are listed, several candidate methodologies are examined for suitability. The candidate methodologies are:

- Data Flow Diagrams (DFD)
- Structured Analysis and Design Technique (SADT)
- Ward-Mellor DFD Extensions
- Hatley-Pirbhai DFD Extensions

The first two are popular requirements specification notations that are familiar in the AFIT environment. The Ward-Mellor and Hatley-Pirbhai DFD extensions are also

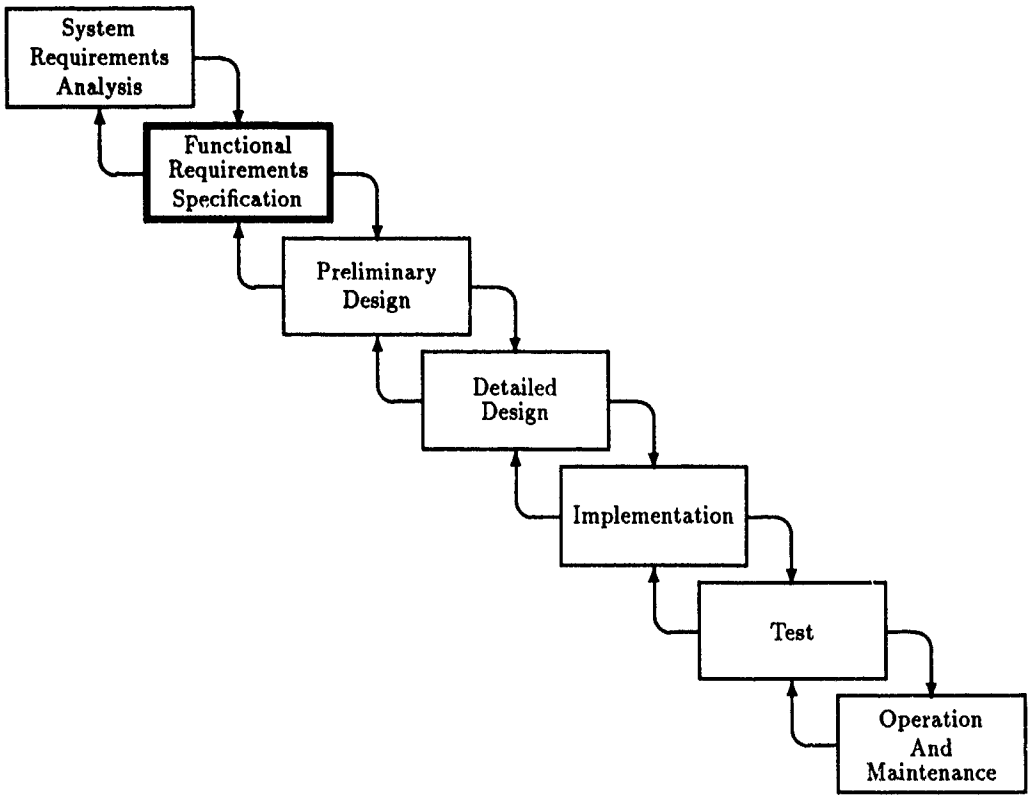


Figure 4.1. Requirements Specification in Software Development Life Cycle (8)

considered because of the special emphasis they place on design for real-time systems. Once advantages and disadvantages of each method have been considered, the SADT notation is selected.

4.1.1 Selection Criteria Many methods exist for specifying software (and other types of system) requirements. To choose one over the others, specific criteria must be identified to guide the selection process. These criteria spell out the important characteristics of the methodology, as they relate to the given project.

(5, 23, 32, 62, 90, 96) provide detailed discussions of the requirements specification process. These sources describe important characteristics of a methodology for specifying the functional requirements of a system. A viable methodology should:

- Provide a graphical representation of the requirements.

- Separate the requirements (the "what") from the implementation (the "how").
- Completely and unambiguously describe the inputs, outputs, functions, and data required.
- Be technically feasible and consistent with the system to be developed.
- Provide a means of iteratively decomposing each high level requirement into its lower level components.
- Be well known and understood to enhance the usability and maintainability of the results.
- Be available in the form of automated tools.

The last criteria was added with the realization that few if any software projects go from analysis to operation without changes to the original specification. As development proceeds, the requirements evolve or become better focused, new requirements are identified, or errors are discovered. If the specification is produced using an automated tool, the designer (and maintainer) of the end product have much more flexibility for changing or updating the specification documents. Further, automated tools often provide additional development support in the form of consistency checks, databases, and so on. While the availability of automated tool support for the selected methodology is not required, it is encouraged.

4.1.2 Methodologies Considered A set of criteria now exists to guide the selection of a specification methodology for the ARTMOS. This section presents a brief description of the four candidate methodologies. Two, Data Flow Diagrams and Structured Analysis and Design Technique, are well know in the AFIT environment. The last two, proposed by Ward-Mellor and Hatley-Pirbhai, modify the DFD approach with extensions for real-time systems.

Data Flow Diagrams (DFDs) (32, 96, 98) are a widely used structured development tool due to their simplicity and understandability. DFD is a graphical representation which emphasizes the data flow (and not control flow) of entire processes. The DFD notation consists primarily of processes (circles) and their interrelations (arrows). This notation can be used to build a hierarchy of processes, giving the software developer the ability to decompose a process into its component processes. Figure 4.2 shows the available symbols.

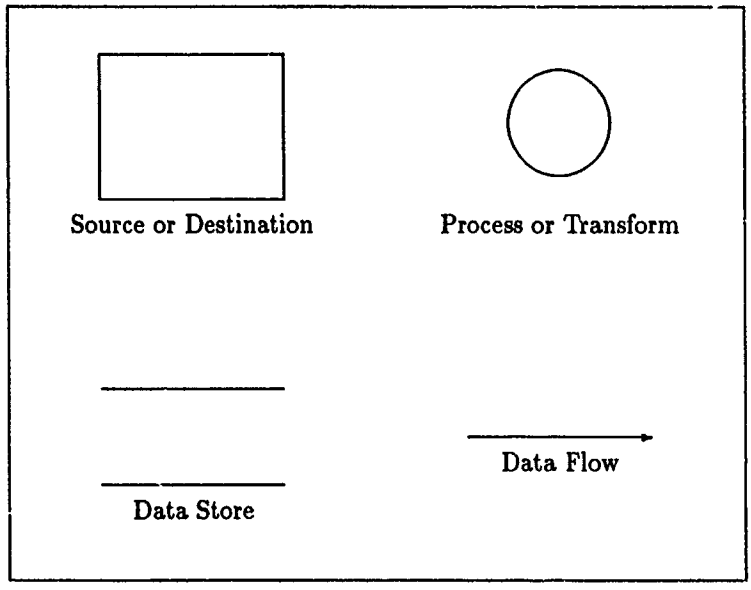


Figure 4.2. Data Flow Diagram Symbols (96:39)

Structured Analysis and Design Technique (SADT) (23, 28, 62) is a tool for the structured, hierarchical decomposition of complex problems. SADT is based on a series of *function diagrams,* where "each function diagram illustrates graphically one level of the decomposition" (28:3). The diagrams consist of boxes (activities) connected by arrows (interfaces between boxes). Each box can be further decomposed as a separate diagram defining the component parts of the original box. The SADT notation has a strict syntax controlling the consistency between diagrams and activities. The arrows connecting activities can be inputs, outputs, controls, or mechanisms.

Ward and Mellor (90, 96) noted that time critical systems tend to be both data and control driven, requiring a model that allows for control as well as data. Traditional specification techniques, such as DFDs, defer consideration of timing and control until the design phase. In real-time systems, these issues cannot be deferred. The Ward-Mellor notation extends DFDs to allow representation of timing and control aspects of real-time systems and their data transformations. The extensions, shown in Figure 4.3, add representations for control transforms, continuous data flow, control flow, and control storage.

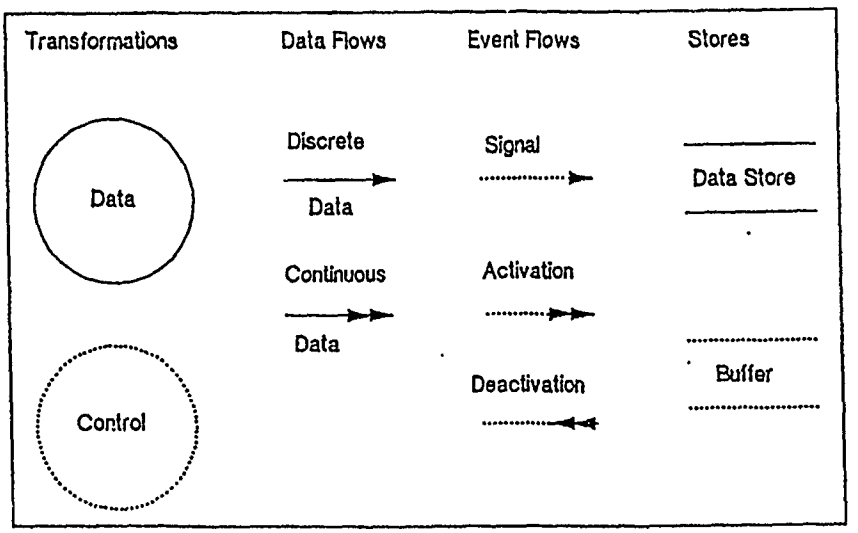


Figure 4.3. Ward-Mellor's Extended DFD Symbols (96:39)

Control flows, *events* in Ward's notation, are data flows with no content, which simply act as signals, and may be used for process activation and deactivation. State transition diagrams specify the control transformations.

Hatley and Pirbhai (29, 96) took a different approach to adding control and timing aspects to DFDs. Unlike Ward and Mellor, who use one model integrating data and control flow requirements, Hatley and Pirbhai use two models, one for functional and one for control behavior. The software developer builds a requirements specification based on distinct data flow diagrams and control flow diagrams at each level (29). This model is different from DFDs, as it shows that the control and timing requirements are as complex and important as those for data processing. The CFDs are specified by state transition diagrams.

4.1.3 Methodology Selection Data Flow Diagrams, as mentioned above, are widely used and an excellent analysis tool because of their simplicity and understandability. Though familiar in the AFIT environment, DFDs focus on data flow and not control flow. There is no clear, unambiguous method for describing the flow of control information through the system. Much of the ARTMOS and its interaction with application software

consists of the passing of control between different activities. DFDs cannot adequately meet this need.

Ward-Mellor corrects this deficiency by adding a number of extensions to the DFD notation. The Ward-Mellor notation allows the developer to represent continuous and discrete data flow, control flow, and data and control stores. Wisotsky (96), however, notes that the essential difference between data-driven and event-driven systems can be illustrated with a single addition, a control flow arrow. This solution represents the time critical nature of the system without cluttering the notation with differences that should be deferred to later in the design cycle. Indeed, the ARTMOS requirements call for control and data flow, but do not need to differentiate between continuous or discrete data, control or data transforms, and so forth. While the Ward-Mellor approach would work, it is much more powerful than needed. Ward-Mellor is rarely used in the AFIT environment, and there are no automated support tools available at AFIT. Note, however, that the development of applications might need the flexibility of this notation.

Hatley-Pirbhai address the lack of control representation in DFDs by adding a distinct control flow diagram to supplement the DFD at each level of decomposition. With this approach, two separate diagrams are needed to fully specify each level. While this is not necessarily bad, it does reduce the simplicity and effectiveness of the methodology as a communication medium: the analyst or programmer has to look in different locations for all the information about one component of the system. As with the Ward-Mellor methodology, Hatley-Pirbhai is rarely used in the AFIT environment, and no automated support tools are available at AFIT.

Of the four candidate methodologies, SADT seems the most appropriate for the ARTMOS development. SADT provides mechanisms for representing both data and control flow clearly and consistently. It is a highly structured methodology with strict syntax, providing an ordered means of hierarchically decomposing a system from high level requirements to low level components. Further, SADT is familiar at AFIT, as it is taught in the Software Engineering courses and supported by an automated tool on the School of Engineering Computer Network. SADT is used to functionally specify the ARTMOS.

4.2 Description of Specification Method

This section presents a brief overview of the methodology that is used to produce the ARTMOS functional requirements specification, SADT. The focus of this description is not to explain every aspect of SADT; rather, emphasis is placed on introducing the notation and symbols used, along with a brief description of the way they can be used to decompose a complex problem. For readers unfamiliar with this methodology, Ross (62) and Hartrum (28) are good sources of information and examples.

The structured analysis (SA) model, as discussed by Ross, considers every important thought about a single important subject (62:99). To achieve this level of detail in a clear and organized manner, the model provides a means of decomposing complex problems into smaller pieces that can be more easily understood. SA begins with a single function representing the system or problem to be modeled. This function is broken down into its major components. These components are in turn broken into *their* major pieces, and so on. Successive decomposition of each function into its major subfunctions continues until no further decomposition is required for complete understanding. The product of SA, called an SA *model,* is a "hierarchically organized structure of separate diagrams, each of which exposes only a limited part of the subject to view, so that even very complex subjects can be understood" (62:97). Strict syntax helps maintain consistency between the levels of the hierarchy. Figure 4.4 illustrates the structured decomposition resulting from proper SA analysis.

The decomposition of a problem is represented by a series of *function diagrams* and associated *facing page text* which provides additional information to supplement the diagrams. Each function diagram depicts one level of the hierarchy, such that "at any given level, a function diagram represents a single function of the next higher level, and presents the major subfunctions of that parent function, along with the interfaces between those subfunctions" (28:7). A function diagram, then, consists of *functions* and *interfaces,* represented by arrows between the functions.

A function, as the name implies, defines some process or activity that must take place, and can be identified by a name starting with a verb, such as "send message."

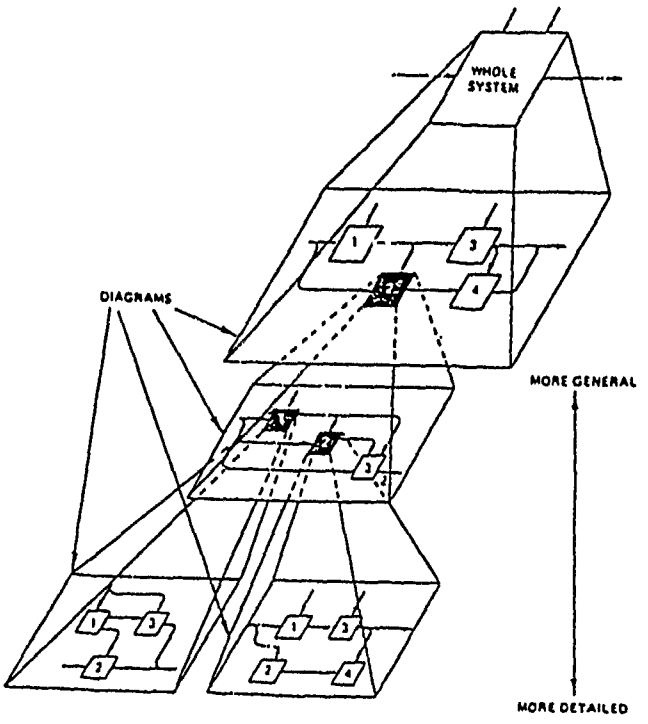


Figure 4.4. SA Hierarchical Decomposition (62:98)

Functions are represented in function diagrams by rectangular boxes. These boxes are labeled with the function name and an activity number. To prevent too much information from being presented in one diagram, a diagram should contain no more than six function boxes (62:98).

Interfaces are shown by arrows entering or leaving a function box. These arrows represent data or information produced by or needed by a function. Figure 4.5 shows that input arrows enter the left side of the box, control arrows enter the top of the box, output arrows exit the right side of the box, and mechanism arrows enter or exit the bottom of the box. "The function is viewed as transforming its inputs into outputs under the guidance of its controls" (28:7) Mechanism arrows provide a means of accomplishing the function, if entering the box, or show that the function is decomposed elsewhere, if exiting the box. Because the arrows entering a box contain data or information needed by the function, the arrows *constrain* the function: the function cannot begin until its inputs, controls, and mechanisms are present. Note, however, that a function might not need all its inputs and

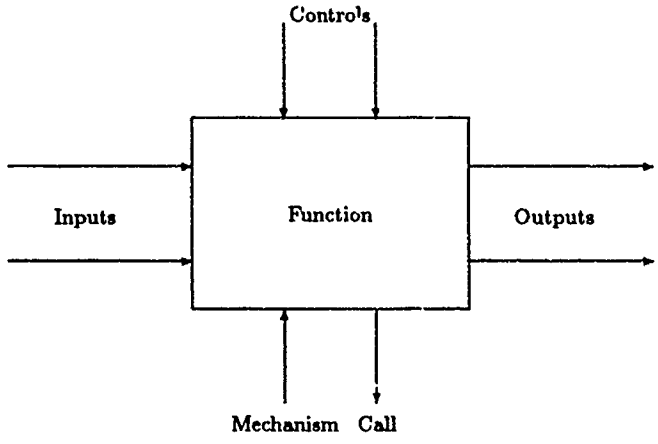


Figure 4.5. SADT Functional Activity (28:8)

controls at the same time. Different components of a function might need only parts of the inputs and control to proceed.

ICOM codes help ensure that a diagram describes the exact same function as the parent function on the higher level diagram. The codes provide a means of matching arrows between a box and its lower level decomposition, which helps maintain consistency and completeness between levels. Each arrow connected to the parent function box is labelled as *I*nput, *C*ontrol, *O*utput, or *M*echanism with a number telling which of the arrows on that side it is. For example, the second input to the parent box would be I2, the third control, C3. ICOM codes refer only to the parent box of the diagram on which they appear, not to the original source of the arrow. Figure 4.6 illustrates the use of ICOM codes.

4.3 Requirements Specification

The complete requirements specification document, with supporting text, can be found in Appendix A. An index of the SADT nodes is given at the start of Appendix A. In this section, the main decisions, alternatives considered, and problems encountered during the specification are discussed. Topics examined include: the target ARTMOS environ-

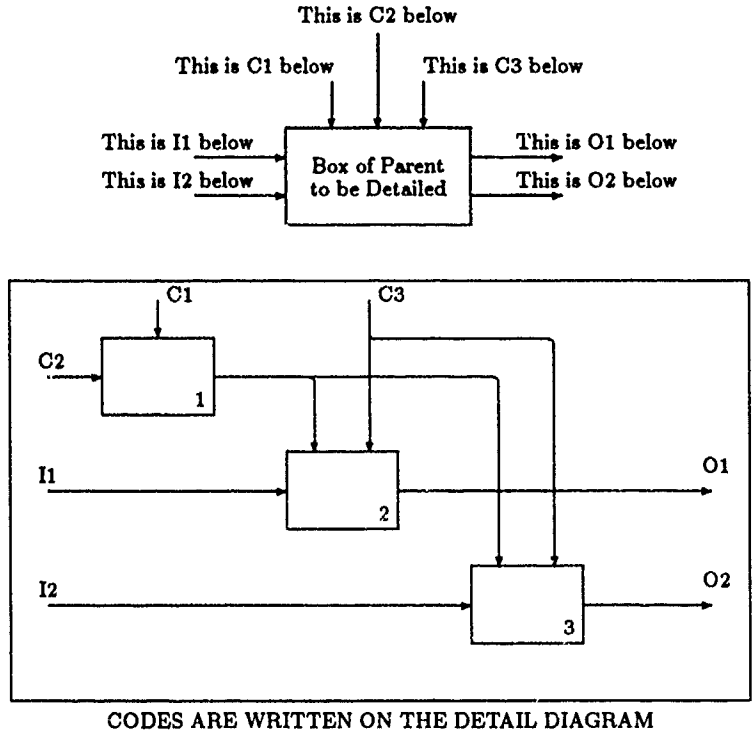


Figure 4.6. SADT ICOM Codes (27)

ment, ARTMOS invocation, structure of the ARTMOS kernel, task and communications management, clock synchronization, and error handling.

4.3.1 The ARTMOS Environment This subsection considers the environment in which the ARTMOS will operate and the manner in which the ARTMOS interacts with that environment. The interactions are shown in Figure 4.7.

The four major components of the overall system are the ARTMOS, the computer hardware executing the ARTMOS, the application software, and the controlled system. The controlled system is the specific application system of interest. In the case of flight control, the airframe (with associated sensor and actuator devices) is the controlled system. Application software performs the actual control of the controlled system. Following the above example, the flight control software receives aircraft status and command information from sensor and communications devices, transforms this information using application

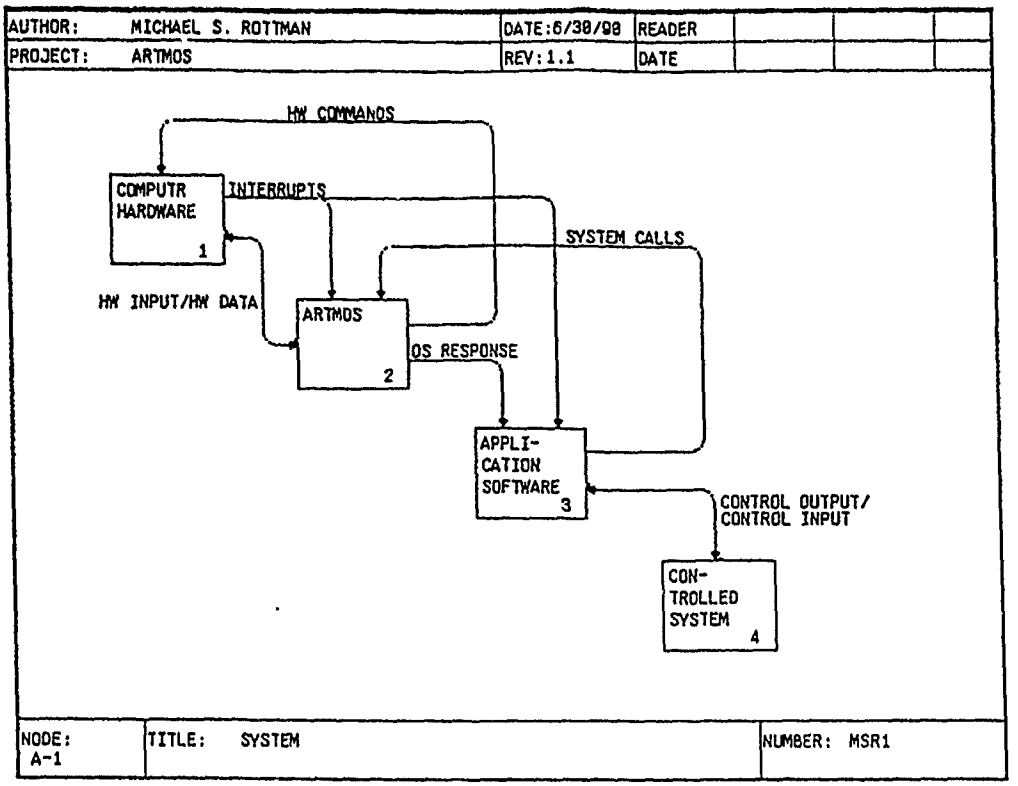


Figure 4.7. The ARTMOS and Its Environment

specific control laws and programs, and outputs the results to the actuators. The ARTMOS is system-level software that provides the interface between the application software and the computer hardware hosting the application and ARTMOS.

The application software requests system services by issuing *system calls* to the operating system. ARTMOS responds by performing the actions requested by the system call and returning any requested information to the application. Output to and input from the controlled system is handled exclusively by the application software; the ARTMOS does not interact with the controlled system at all. Interrupts from the *computer* hardware are handled entirely by the ARTMOS, unless the application software provides replacement interrupt handlers during system initialization to enable different actions in response tothe interrupts. Interaction between the computer hardware and the ARTMOS consists of hardware interrupts and input provided to the ARTMOS, with commands and data issued to the hardware in response.

For an actual application, both the computer hardware and the controlled system are distinct entities. During ARTMOS development, however, the software will be running on a laboratory development system, not on a computer embedded in the controlled system. For test purposes, the controlled system will have to be simulated by the development computer or by an external computer.

4.3.2 ARTMOS Invocation An important issue in the development and use of the AFIT Real-time Multiprocessor Operating System is that of invocation. How will the ARTMOS be started? In a true embedded system, this issue is not a factor. When the system is powered up, the computer begins operation at a pre-defined location in the OS or application software. The software need not be explicitly loaded, for it is already resident in read-only memory (ROM). In some applications, the OS code begins by loading application software to random access memory (RAM) from some external storage, allowing mission software to be modified between runs without replacing the ROMs. Alternately, all software might be ROM resident, so that no download of code would be needed.

ARTMOS, however, is not intended for a "real" embedded system in this first phase of development and demonstration. The initial implementation targets a laboratory development system, with plans to later port the ARTMOS to an actual embedded system. For the development and test phases, when many corrections and modifications are likely, it would not be practical to require the OS to execute from ROM. A laboratory system can have the added advantage of development tools, such as compilers, linkers, loaders, and debuggers which may be invaluable.

In a non-embedded system, as described above for the initial ARTMOS development, some capability is needed to load the OS and application code. Depending on the specific target system chosen, additional code may or may not have to be developed to handle this requirement. On the Intel iPSC Hypercube, for example, the cube manager has facilities to load the OS and application software to the nodes. For a VME card cage, with no resident OS, a boot monitor might be needed.

It should be noted that the software to load the OS is not explicitly part of the operating system. The loader is really a separate entity used to invoke the OS. However,

since this capability is essential to making the ARTMOS usable, this issue must be considered. For this reason, the ARTMOS specification begins by showing the loading of the OS and application code. It must be noted, however, that this activity cannot be specifically defined without details of the target hardware, which are not considered until the design phase. The diagram shows that the OS must be loaded somehow, but cannot describe how this is accomplished until later in the development process.

4.3.3 Structure of EXECUTE OS The structure of the requirements specification requires some explanation, particularly for the main body of the ARTMOS, EXECUTE OS (node A3). A number of approaches exist for specifying the execution requirements of the ARTMOS. After several attempts, refinements, and rejections, one particular approach was selected for this specification. This section describes the rationale behind this structure.

As Figure 4.8 shows, the main body of the ARTMOS specification is broken into three major activities: EXECUTE TASK, HANDLE SYSTEM CALLS, and HANDLE INTERRUPTS.

The requirements analysis emphasizes the definition and description of the functions the ARTMOS needs to perform. The initial attempt at graphically specifying these requirements focused on the functional areas described during the analysis: task management, communications management, time management, and so on. This attempt fell short because it could not adequately describe the manner in which the different types of activities, system calls and interrupts, were invoked. With this approach, activities were grouped together because they dealt with the same functional area, despite the fact that they were invoked in different ways at different times and performed different actions.

The second attempt considered both the manner in which the functions interact and the different ways that they can be invoked. This attempt broke out separate activities to handle system calls and interrupts, as these are the only two ways to get the operating system's attention. When the OS receives a system call or an interrupt, it performs some action or series of actions in response.

After some examination, however, an additional activity was broken out at this level, that of executing a task. The purpose of the multitasking kernel is to switch between

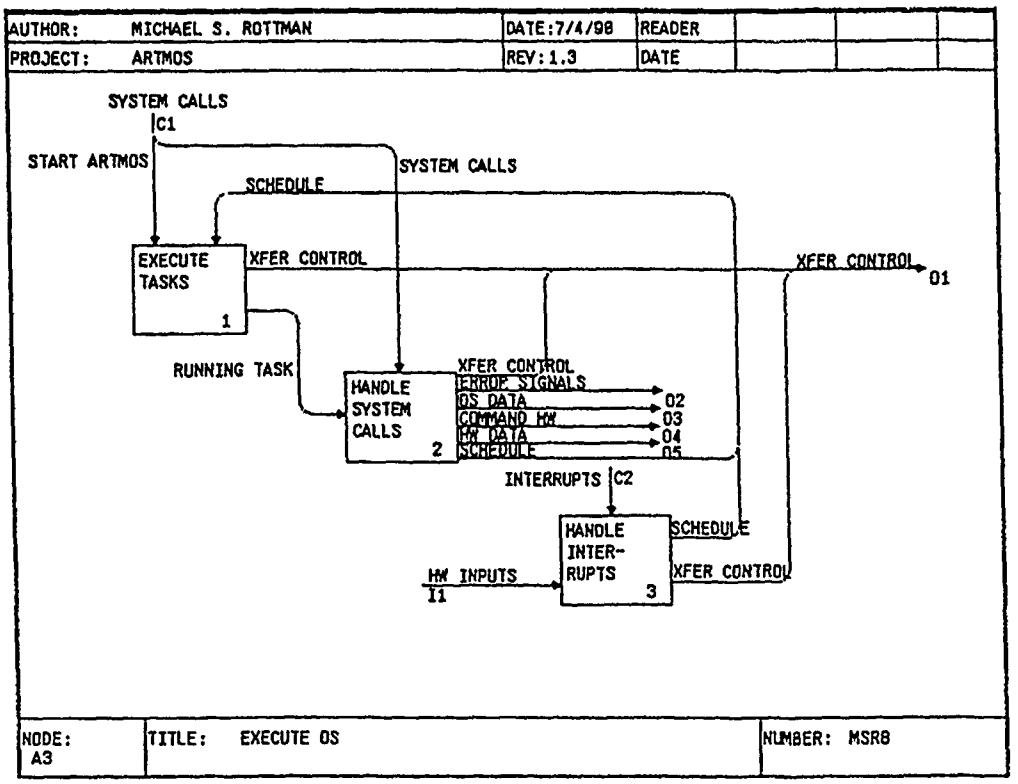


Figure 4.8. EXECUTE OS SADT Diagram

application tasks on the basis of some task prioritization and some scheduling algorithm. Though the scheduling of tasks occurs as a result of system call or interrupt activities, it is a functionally independent activity and as the main element of the kernel should be emphasized.

4.3.4 Task Management The purpose of this section is to explain some of the decisions made in the specification of the task management functions. Section 3.4.1 presented an overview of the task requirements and examined the four states a task may occupy. This conceptual discussion examined what the ARTMOS must do functionally without considering or assuming how these functions are implemented.

As also discussed in Section3.4.1, the tasks are not physical entities that may be manipulated by the OS. Rather, each task is represented in the OS by some form of Task Descriptor or Task Control Block (TCB). TCBs serve as a place to store the state of the

tasks when they are not in the *running* state. The task state is information defining the exact status of the system when the task finished executing previously, so that the task continues to operate as though never interrupted. While the requirements specify that some data structure (a TCB) is used by the OS to manipulate and manage tasks, they do not define the exact nature of the data structure or how it is manipulated. These questions are more appropriately answered in the design phase. It is sufficient at this point to merely specify the existence of these structures.

Likewise, it is not a design issue to identify the requirement for some type of "list" of ready or blocked tasks. Such a requirement does not specify the exact nature of the list, nor does it place any constraints or assumptions on the TCB data structure. This requirement merely states that some data structure is needed to manage the generic TCBs as the tasks pass through the various states. Selection of a particular implementation of this specification takes place in the design phases.

It is the responsibility of SADT node A31, EXECUTE TASKS, to perform the scheduling and dispatching of applications tasks over time. Figure 4.9 shows this activity. When a system call or interrupt handling activity performs actions which alter or could have altered the set of tasks ready for execution, EXECUTE TASKS is triggered to dispatch the highest priority ready task. In some cases, the running task has released the processor and another task must be scheduled. An example of this is the TERMINATE TASK system call, which kills the running task. A different ready task is selected and dispatched. In other cases, however, the set of ready tasks has been altered and EXECUTE TASKS must determine if a task of higher priority than the running task is now ready. If so, the running task is preempted and the new task is scheduled. The clock interrupt handler, for example, increments the clock value and wakes any tasks delaying until the new time. The new ready task(s) may or may not be of higher priority than the running task.

4.3.5 Memory Management The original requirements in Section 3.1.1 give the application the ability to manage memory through system calls for getting and releasing a block of memory. The motivation for this requirement was that each processor had to have

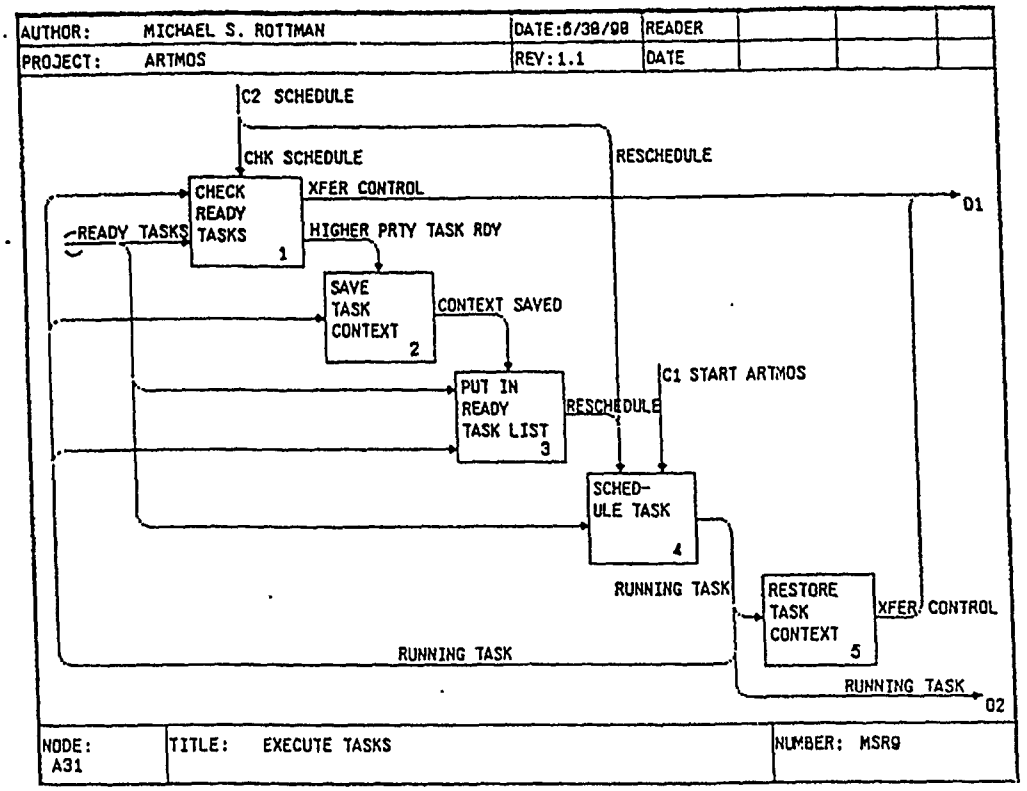


Figure 4.9. EXECUTE TASKS SADT Diagram

at least enough memory for the maximum task load anticipated for that processor. Tasks would be able to request or release memory as needed without the possibility of running out of available memory.

However, this capability adds a great deal of overhead to the operation of the OS and application. When a task begins operation and needs memory, it performs a GET MEMORY system calls to acquire the necessary memory space. When it no longer needs the memory, the block can be released to other tasks that might need it. The cost of this approach is paid at runtime, when there may or may not be enough time available to perform the memory operation.

Consider the advantages of a task being able request memory when it is needed and release it when it is no longer necessary. The primary advantage is that a task only possesses a block of memory when it is needed. If a periodic task is through processing for the moment, it can release its memory to another task which might need it. This approach

is economical in that the resource, memory, is only claimed when it is needed and available when not needed.

One ARTMOS requirement derived in Section 3.4.4, however, is that each processor must have sufficient memory for its maximum projected task load. Rationale for this requirement is that a flight or life critical real-time system cannot afford to wait for an essential resource (such as memory) which may or may not become available soon enough. Because each processor by definition has "enough" memory, there is no benefit gained by allowing the tasks to dynamically request memory as needed. Rather, the necessary memory can be allocated when the task is created, substantially reducing the amount of runtime overhead. The price of the new requirement is paid during initialization, which takes longer because of the memory allocation overhead. Memory is deallocated when tasks are terminated.

This new memory management requirement appears in the SADT diagrams in two locations: the ASSIGN MEMORY (A2243) and RELEASE MEMORY activities (A32111). These diagrams show that the memory management functions are operating system activities rather than system calls, as originally specified.

4.3.6 Communications The specification of the ARTMOS communications requirements is by necessity somewhat ambiguous and requires an explanation. While the requirements are specific as to what communications activities must occur in the ARTMOS, these requirements cannot be clearly specified because of the dependence of communications upon the target hardware. This issue is examined in the following discussion.

The requirements developed in Section 3.4.2 call for a synchronous receive operation and both synchronous and asynchronous send operations. The synchronous or asynchronous nature of the operations is based on the use of a producer/consumer algorithm to protect the integrity of data exchanges. A receive operation, for example, is dependent on the desired message being available to be received (the message has been "produced"). If the message has not yet been produced, the task waits until either the message arrives or a specified amount of time has passed. The receive is "synchronous" because it uses the producer/consumer algorithm to know when the desired message is available. Send operations

can be synchronous also, if they use the producer/consumer algorithm. In this case, a task attempting to send a message waits for a specified amount of time if the previous message to the destination mailbox has not yet been consumed. A send is asynchronous, however, if it sends the message regardless of whether the previous message has been consumed yet.

These requirements seem straightforward. A receiving task consumes the message if it is available and waits otherwise. The ambiguity rises from the many ways one can specify the requirement to "wait if the message is not available." Does the task go to sleep on some message wait queue, to be awakened when the message arrives? Does the task poll on the mailbox status, continually checking the status until the message arrives or a timeout occurs (thereby blocking all other tasks)? Does the task poll, but in such a manner that if it checks the status and the message is not available, it releases the processor? Each time the task is switched back into the CPU, it checks the mailbox again. These are all design questions.

What, then, must be specified? The essential requirements are 1) the task waits for an interval if the message has not yet been produced, and 2) the task releases the CPU while it is waiting. The second requirement is derived from the real-time nature of the application: life or flight critical deadlines might be missed if tasks are blocked by a task that is "busy waiting." The question, then, is how to specify these two requirements?

Several unsatisfactory iterations led to the conclusion that there may not be an implementation-independent method of specifying the requirements. The specification could show a task "blocking" if the message is not available. When the datagram is received at the processor and the message posted to the mailbox, the task could be "unblocked" and given the message. While this approach seems suitably implementation or hardware independent, there is no requirement for the OS on a processor to be notified when messages arrive at the mailboxes. With some systems, it may not be feasible or even possible to provide this capability. Likewise, there are no explicit requirements to poll on the mailbox status. The point is that the communications operations depends heavily on the target hardware and the particular needs of the system. There may not be a generic specification of the communications requirements.

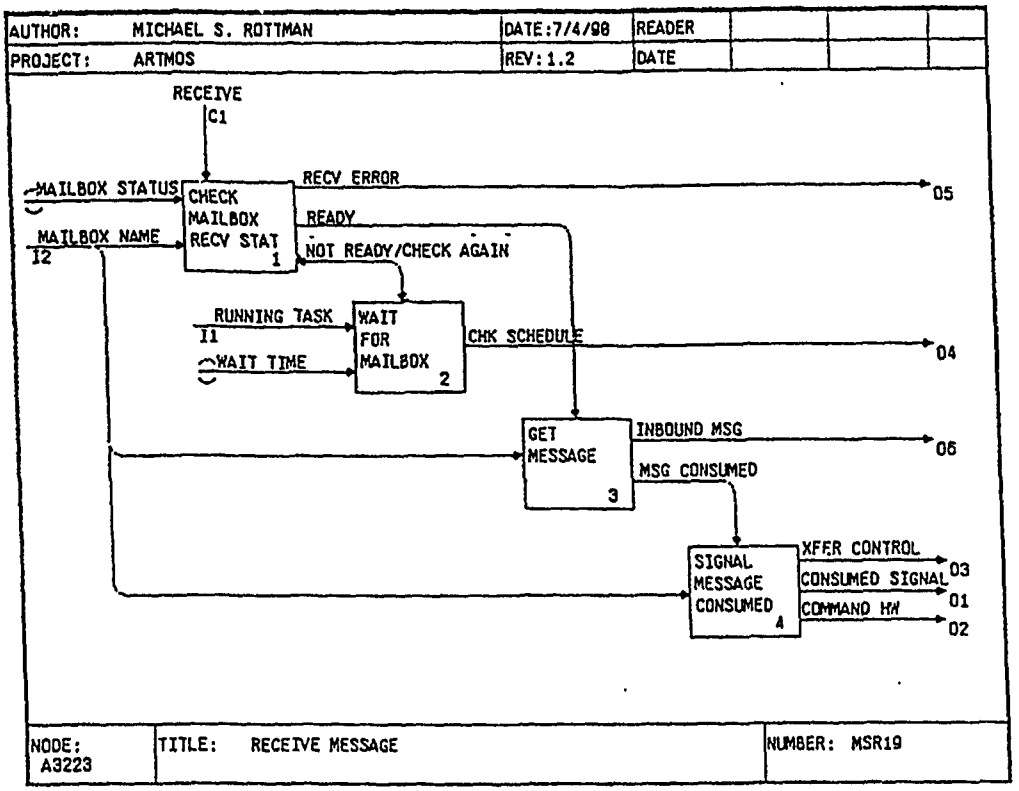


Figure 4.10. RECEIVE MESSAGE SADT Diagram

The specification diagram developed for ARTMOS communications, demonstrated with the RECEIVE MESSAGE operation in Figure 4.10, is a concession to the above points. The diagram breaks the process of blocking for a message into two activities: checking the mailbox status and waiting for the mailbox status to change. If the message is not ready, the task waits. At some point, the mailbox is checked again. If the message is still not available, an error signal is passed back to the application task. To meet the second requirement above, a rescheduling operation is triggered to allow other tasks to proceed. With this framework, the implementation can do polling or actually block the task on a separate list until the message arrives or a timeout occurs, or whatever other approach is appropriate for the system hardware. Specific details of this process are inappropriate for the specification phase, and must wait until the design phase. When a message has been successfully received, a *message consumed* signal is sent to update the producer/consumer algorithm.

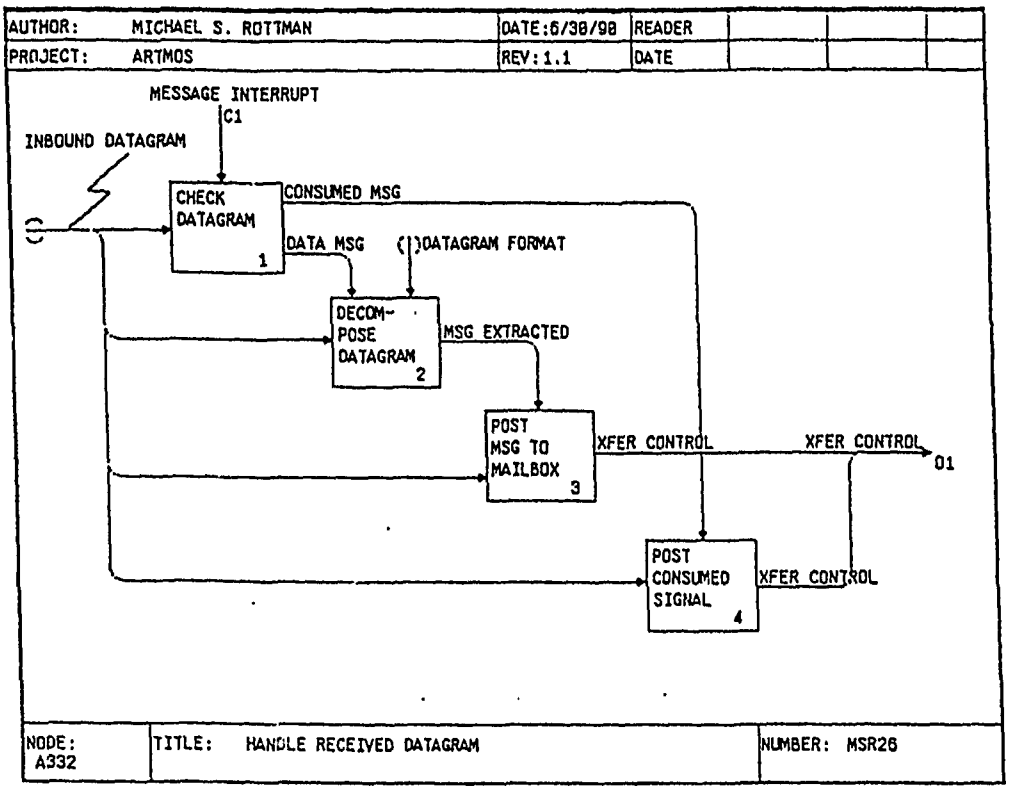


Figure 4.11. HANDLE RECEIVED DATAGRAM SADT Diagram

The final element missing from the communications picture relates to how messages are actually received and how (if at all) the concerned tasks are notified of their messages' arrival. Figure 4.11 shows this process, HANDLE RECEIVED DATAGRAM, node A332. This diagram depicts the process of extracting a data message from a received datagram and posting it to the destination mailbox. Likewise, a consumed signal is posted to the appropriate mailbox. In neither case is a task waiting for the datagram awakened. The reason for this is as follows.

There is no explicit requirement for an interrupt to handle received datagrams; it is drawn as such, however, to bring out the fact that *something* has to perform these activities. Because this activity is asynchronous and occurs on receipt of the message by the hardware, an interrupt seems the most appropriate way to display this requirement. In some systems, this activity might be accomplished entirely in hardware transparent to the software; in others, the OS might be entirely responsible for processing the received datagram. If

processing of the datagram is handled by the hardware without OS intervention (as in the AMCAD system), there is no way to signal the blocked tasks. For this reason, this action was not specified.

4.3.7 Clock Synchronization Synchronization of the different processor clocks is another area where the final implementation is intimately tied to the particular target hardware. The requirements derived in Section 3.4.6 stated that the processor clocks had to remain within a given range, or window, of one another. While this is fairly straightforward, the specification can easily become cluttered if design issues are considered to soon. The specification, shown in Figure 4.12, describes only the requirements that the clocks be synchronized and that any blocked or waiting tasks be checked in case the changed time has altered the task's status. Questions of *how* the clocks are synchronized must be deferred until later in the ARTMOS development.

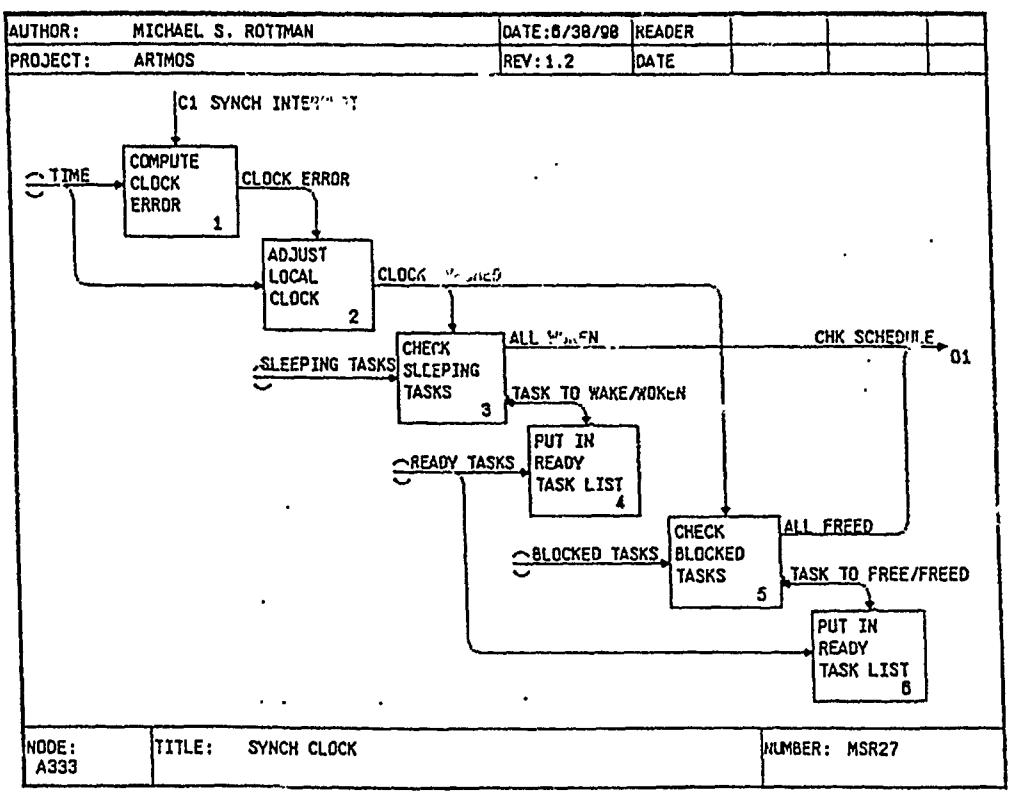


Figure 4.12. SYNCH CLOCK SADT Diagram

As above, there is no explicit requirement that the processor be interrupted when it is time to synchronize. The synchronization requirement is specified this way to emphasize the fact that synchronization must occur periodically. As with the reception of datagrams, clock synchronization could, for some systems, be implemented entirely in hardware. If the system has a fault-tolerant clock mechanism, only one clock is used by the processors so no synchronization is needed. If, on the other hand, each processor has a completely independent clock, they must be synchronized in some way to make sure the processors agree on the system time. Hardware circuits might be available to transparently synchronize the clocks, or software algorithms might be needed. The specification merely describes the required actions.

4.3.8 Error Handling This subsection considers the important topic of error handling. The requirements developed in Section 3.4 made no mention of error handling due to an unclear understanding of the role of the ARTMOS in error cases. During the specification, several situations arose in which it was unclear which software entity was responsible for handling error conditions. Was the OS expected to attempt recovery from the error, or should that be the responsibility of the application?

If the ARTMOS is assumed to be "correct," two classes of errors can be identified: errors caused by incorrect application execution or caused by operation of the system. In the first case, it is possible that the application software itself is erroneous. For example, the application might incorrectly attempt to start the OS before it has triggered OS initialization, or might not initialize critical data structures. With errors of this nature, the ARTMOS is pretty much helpless. It is physically impossible to anticipate all the myriad possible ways in which an application could function improperly, and even more difficult to devise OS corrections for these errors. The application must be responsible for its own actions, and the application designer must be responsible for adequately testing and/or verifying his design.

The second class of error derives from operation of the entire system: application and hardware together. Types of errors in this class include messages that don't arrive (or not in time), data that is never consumed, and attempting to delay until a time that

is already past. It is difficult to identify whether these errors stem from hardware faults, bugs in the application code, unsynchronized clocks, or missed deadlines. For these errors, what happens? If a task is waiting for a message that does not arrive in time, for example, does the OS ask the sending task to repeat its transmission? Does the OS simply notify the receiving task that its message is unavailable, letting the application ask for a re-send?

In answering these questions, two issues were considered. First, error handling can significantly slow down the functioning of the OS, extending the OS overhead and delay before control is returned to the application tasks. Small (69) notes that real-time operating systems cannot afford a great deal of error handling capability, lest they become too slow and undeterministic to satisfy the requirements of real-time applications. Secondly, different applications or even different circumstances within the same application may require different treatment of the same error conditions. Rather than radically change the ARTMOS to meet the error handling requirements of each application, it was decided to have the application handle its own error corrections. It is the responsibility of the OS to notify the application when specific errors are identified.

4.4 Chapter Summary

This chapter discusses the development of the functional requirements specification for the AFIT Real-Time Multiprocessor Operating System, ARTMOS. The specifications illustrate what functions the software must perform to meet the requirements presented in Chapter 3. The development begins by examining the criteria for selecting an appropriate methodology for specifying the ARTMOS. Next, several methodologies are considered with respect to the defined criteria. The method chosen, SADT diagrams, is then examined more closely to familiarize the reader with SADT notation and techniques. Finally, specific elements of the final specification are examined to show the decisions made and alternatives considered. Appendix A contains the SADT diagrams and facing page text, with a node index. Specific diagrams from the specification are included in this chapter for clarity or as examples.

The next chapter transforms the functional specification developed here into a preliminary design of the ARTMOS.

V. Preliminary Design

This chapter describes the preliminary design phase of the ARTMOS development, as shown in Figure 5.1. Section 4.4 develops a functional requirements specification for ARTMOS defining functionally *what* the software should do. The process of transforming the *what* must be done into *how* it will be done is known as design. "Design is the process of applying various techniques and principles for the purpose of defining a device, process, or system in sufficient detail to permit its physical realization" (58:128). The design process, however, can be further decomposed into *preliminary* and *detailed* design phases. Preliminary design is concerned with defining the necessary software structure to provide the specified functions. The major components of the software are identified, along with the relationships between the components. The emphasis is on the function each module is to perform, not how the module performs the function. Major data structures are identified and described. The output of this phase is a high-level model of how the system accomplishs its task, but without sufficient detail to implement (55:12). Detailed design refines the preliminary design to the point where implementation can begin. The modules identified in preliminary design are further decomposed, module interfaces are detailed, and specific data structures and algorithms are designed. Hardware selection occurs, if a specific target system is not required. Detailed design produces a blueprint containing the information needed to code the software (58:130).

This chapter examines the preliminary ARTMOS design. It first considers several design methodologies in order to select an appropriate one. The selected method is described briefly to familiarize the reader with the notation. Finally, key elements of the design are examined to show the alternatives considered, problems encountered, and decisions made.

5.1 Design Methodologies

According to Pressman, a design methodology should show hierarchical organization, lead to modules exhibiting independent functional characteristics, and be derived from repeatable methods driven by information from the requirements analysis (58:128). Many

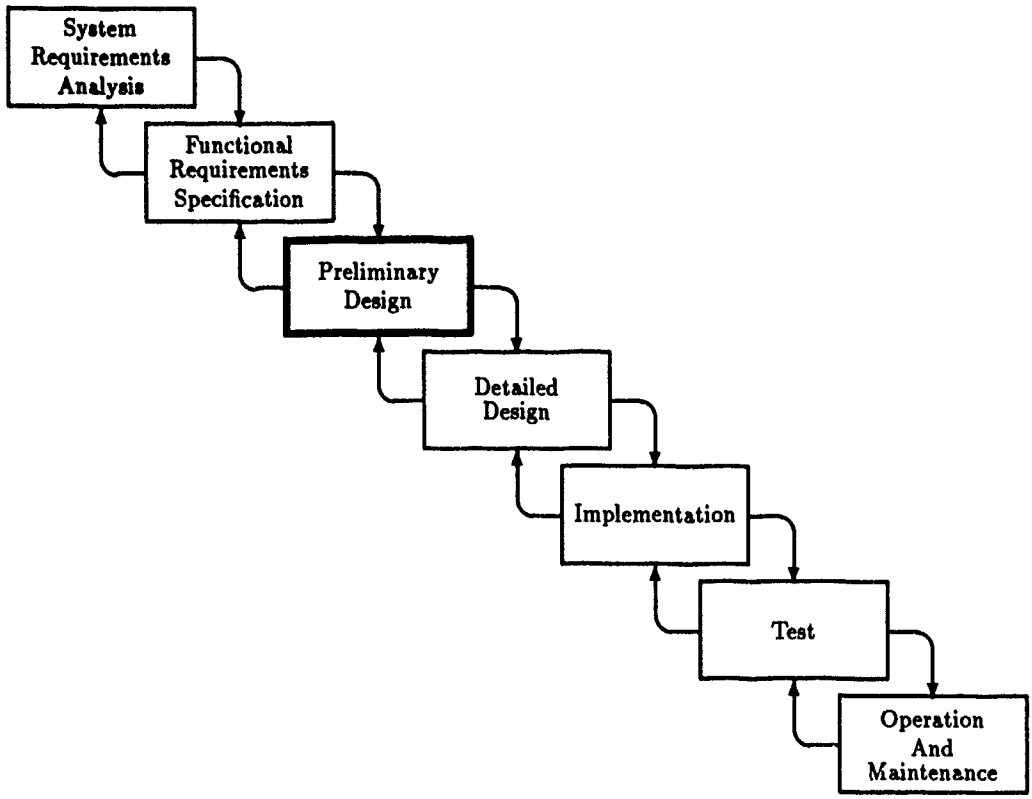


Figure 5.1. Preliminary Design in Software Development Life Cycle (8)

different methods are available that meet these criteria. Perhaps the most widely used is Structured Design, developed by Yourdon and Constantine (97).

Other methodologies explicitly focus on the requirements of real-time systems. As described in Section 2.1, real-time systems differ from non-real-time systems in that many different activities must occur asynchronously to accommodate different speed hardware devices and other independent deadlines imposed by the system environment. The activities, or *processes*, often must communicate or synchronize to meet sequential dependencies between processes. A real-time design method must provide means for decomposing the system into concurrent processes and must describe the dependencies between them. Two such methodologies, Design Approach for Real-Time Systems (DARTS) (25, 26), and Layered Virtual Machine/Object Oriented Design (LVM/OOD) (44, 51) are considered in addition to Structured Design.

5.1.1 Structured Design Structured design seeks to conquer the complexity of large systems by partitioning the system into "black boxes" and organizing them into hierarchies (55:11). A black box has a defined function and known inputs and outputs, but the implementation of the box is unknown. Each box is decomposed into its major functions by analysis of its inputs, outputs, and transform. The subfunctions comprise new black boxes which in turn can be further decomposed until the system design is complete. Structured design is often referred to as data flow-oriented design because of the way the functional decomposition is based on the transformation of data in the system.

In structured design, a set of data flow diagrams is developed which models the flow of data through the system and the logical transformations that are made on the data. When the system is decomposed to sufficient depth, the data flow diagrams are converted to structure charts by transform and transaction analysis. Transform analysis identifies the areas in the system that process inputs to produce outputs, defining the afferent (input), transform (process), and efferent (output) flows in each area. Transaction analysis decomposes the system with respect to data items that trigger some action or actions.

One advantage of this scheme is that it provides qualitative measures for evaluating the design: coupling and cohesion (5:305) (55:60). These measures allow the designer to evaluate the "goodness" of a design by considering the specific functionality of each software module and how it interfaces with the rest of the system. "Effective modularity results in minimal connections (low coupling) and maximum module independence (high cohesion)" (32:435).

Bergland notes, however, that partitioning strictly on the basis of afferent and efferent flows is artificial at best (5:305). Placing input and output in separate "ears" may not produce an accurate model of the problem. Structured design is also limited by its inability to explicitly define hardware interfaces and model concurrent operation. It is flexible enough to be used for these purposes, but the methodology does not define a standard way of doing so.

5.1.2 Design Approach for Real-Time Systems (DARTS) To support the asynchronous nature of real-time systems, DARTS extends the conventional structured analysis and design method by providing both "an approach for structuring the system into tasks and a mechanism for defining the interfaces between tasks" (26:658). From the functional specification, the data flow through the system is analyzed to determine the major functions. Data flow diagrams, described in Chapter 4, are used to decompose the system functions until all major subsystems and their components are identified. When the DFDs are developed to sufficient depth, concurrent tasks are defined based on the asynchronous functions. DARTS provides specific criteria for grouping or isolating functional transforms as tasks, until all tasks in the system are identified.

DARTS defines interfaces between tasks with two objects called task communication modules (TCM) and task synchronization modules (TSM). A TCM handles all communications between tasks, and consists of a data structure and all operations on the structure. TCMs support loosely and tightly coupled message passing, as well as shared memory communications. Synchronization between tasks is done with events. Destination tasks can wait for an event, or source tasks can signal an event that activates the destination task. This capability is provided by the TSM.

When all tasks and interfaces are defined, each task is designed. Since tasks themselves are defined as sequential programs, data flow diagrams can be used to decompose them. Structured design is then used to develop structure charts for each task identifying the modules in that task and the interfaces between them. In this manner, the DARTS extension are used to identify the parallel and asynchronous components of the system and each of those components is designed in the traditional way using structured design.

In many real-time systems, whether or not an operation can be performed is affected by the state of the system. A state transition manager may be needed to maintain the current system state and to validate requested actions.

5.1.3 Layered Virtual Machine/Object Oriented Design Nielsen and Schumate's LVM/OOD method (51) was specifically developed for large, real-time distributed systems using the Ada language. It provides capabilities for defining concurrent processes

and the interactions between them, as well as including support for real-time considerations. LVM/OOD is based on the design concepts of layered virtual machines and object oriented design.

LVMs are a means of abstracting algorithms into independent processes that can operate in parallel (44:48). Processes are virtual in that they are independent of the system hardware. Modules are defined hierarchically, "layering," which defers implementation details and supports information hiding. OOD, as described above, abstracts problem "objects" into data structures and operations that act on them, known as object managers.

The LVM/OOD methodology consists of nine steps, which Lemansky decomposes into two distinct phases. The first four steps comprise the system design phase, which roughly corresponds to the preliminary design stage of the software development life cycle. This aspect of the methodology is "ideal for real-time distributed systems because it provides methods for describing hardware interfaces, timing constraints, and concurrent processes" (44:50). The steps of system design are:

1. Develop a context diagram describing the hardware interfaces to the software system.
2. Assign control processes to each of the external devices, called "edge functions." These processes are simple device drivers and should contain the minimum number of instructions needed.
3. When all edges have been defined, the rest of the software — the "middle part" — is decomposed into its major components. Data flow diagrams can be used to model the components and their interactions. Further decomposition may be required for complex components.
4. Abstract the components into independent processes that can run concurrently with a minimum of interaction. Nielsen and Shumate provide guidelines for grouping or isolating components into processes (51:89–91). These guidelines are similar to those for DARTS (25).

System design is complete when all components have been abstracted into processes. The output of this phase is a high-level abstraction of the problem in the form of a set of

concurrent, communicating processes (44:50). Detailed design, the final five steps of the methodology, defines the algorithms and objects needed to implement the system in Ada. To this point, the design is independent of the implementation language. Detailed design focuses specifically on using the Ada language.

1. Describe the interfaces between the processes created in system design. The interfaces define the communications and coupling between the processes. The processes and interfaces are shown using a process structure chart (which is different from the structure charts used for data flow design).
2. Translate the processes into Ada tasks. Intermediate tasks can be added as needed to ease communications and decrease coupling. The result is an Ada task graph.
3. Encapsulate the tasks into Ada packages. Objects used for interprocess communications should be abstracted into data objects and encapsulated into packages.
4. Translate the design into an Ada Program Design Language (PDL). The graphical representations must be translated into programming constructs using an Ada PDL.
5. Large tasks should be further decomposed to their components using the LVM method. This decomposition may introduce another level of concurrency which must itself be defined with process structure charts and task graphs.

5.2 Selection of a Design Method

Structured design is a very popular design methodology and can lead to highly functional modular designs. The use of data flow diagrams to decompose the functional requirements allows the designer to represent the component functions of a system and their interaction. The criteria of coupling and cohesion (32, 55, 58) can be applied to iterate the decomposition and module design to improve the quality of the software. However, structured design cannot explicitly represent hardware interaction, nor does it provide the capability to identify and design concurrent processes, both important issues for real-time software. It also does not address the internal design of the modules (25:939).

The DARTS and LVM/OOD design methodologies have a number of similarities, particularly in the preliminary design stage. Both use DFDs and a similar set of process

selection heuristics to define a set of concurrent processes. A similar graphical notation is used by both to represent process interaction. After the system has been decomposed into abstract concurrent processes, however, the two methodologies diverge in approach. DARTS uses structured design to functionally decompose large or complex processes, while LVM/OOD uses the layered virtual machine approach to develop lower-level objects. Both methodologies utilize finite state machines for state dependent processes. The primary difference between the two methodologies is that, once a set of processes and their interfaces has been defined, LVM/OOD focuses the design on the specific features and capabilities of Ada. (The authors of LVM/OOD briefly discuss application of the methodology for non-Ada implementations, but note that Ada is most suited for the LVM approach (51:228).)

While Ada is a consideration for the ARTMOS implementation, it is not a requirement. Current implementations of Ada compilers and run-time systems are deficient with respect to parallel processing systems, despite the fact that Ada has built-in support for parallel processing in the Ada task. The only truly parallel Ada compilers currently available assume a shared memory architecture, such as an Encore Multimax.

The key point is that the ARTMOS could be implemented in any language. None of the ARTMOS requirements preclude any implementation language. In some cases, different languages may have to be used for different systems, depending on compiler availability. If the functionality and application interface to the ARTMOS remain consistent between implementations (one of the objectives of this investigation), the application is unaffected by the selection of ARTMOS implementation language.

Based on these considerations, it does not seem reasonable to strictly follow a design methodology that is firmly tied to one particular implementation language. Certain useful features of LVM/OOD, however, are not included in the DARTS methodology. Perhaps the best approach is to apply a hybrid design methodology combining the best of DARTS and LVM/OOD. The specific methodology is described in the next subsection.

5.2.1 Description of Selected Method The design methodology that is used for the ARTMOS development is a hybrid of the DARTS and LVM/OOD methods. The approach taken to building this methodology is to take the basic framework of DARTS and supple-

ment it with elements of LVM/OOD. Particular aspects of LVM/OOD that are included are explicit consideration of system hardware requirements early in the design and the encapsulation of processes into packages. Ada packages cannot be used unless Ada is the implementation language, of course, but "packages" can be emulated in other languages through header and "include" files. Packaging supports information hiding and modularity.

An important consideration for this methodology concerns its division into preliminary and detailed design phases. This division point should be a natural transition point in the design process, not an arbitrary break for convenience. Neither DARTS nor LVM/OOD define design phases, though Lemansky (44) shows system and detailed design stages. System design identifies hardware devices and interfaces, decomposes the remaining functionality of the system, then identifies the concurrent processes. The remaining LVM/OOD steps are performed during detailed design. Lemansky bases this division on the fact that the system design is language independent, while the detailed design explicitly addresses the features of Ada.

Since the ARTMOS design defers the selection of an implementation language as long as possible, a different criterion is needed to distinguish between preliminary and detailed design. DARTS and LVM/OOD both provide a natural transition between high- and low-level design activities that define distinct design phases. During preliminary design, the components of the system are abstracted as a set of concurrent processes and the interfaces between them. The detailed design phase performs and designs each individual process and encapsulates the final design into modules.

Preliminary design consists of five steps, the result of which is a set of concurrent processes and their interfaces. The basis for all functional decompositions is the requirements specification outlined in Chapter 4.

1. Develop a context diagram showing the hardware interfaces to the software system. Each of the hardware devices is depicted separately while the internal control system is represented as a single abstract entity.
2. Assign separate control processes to each of the external devices, or edge functions. These are device drivers and contain the bare minimum instructions.

3. When all edges are defined, the "middle part" is decomposed into its major components. Starting with the functional requirements, data flow through the system is analyzed and major functions identified. Data Flow Diagrams (DFD) are developed and decomposed to sufficient depth to identify the major subsystems and their components.

4. Abstract the components into independent processes that can run concurrently with minimal interaction. The DFD transforms are analyzed to identify which may run concurrently and which are sequential in nature. The DFD is redrawn, with boxes around each transform or set of transforms that logically form a processes. Criteria are provided for guiding process identification.

5. Formalize the process interfaces. The interfaces define the communications and coupling between the processes. The processes and interfaces are shown with a process structure chart (different from the structure chart used in structured design). Figure 5.2 shows the types of data flow between processes, which is treated as:

   (a) a loosely coupled message queue if processes pass information and may proceed at different rates.
   (b) a closely coupled message/reply if the sending process cannot proceed until it getting a reply from the receiving process.
   (c) an event signal if only a notification of event occurrence and no data transfer are required.
   (d) a data store needing to be accessed by two or more tasks. Handled by a monitor process defining the data structure, access routines, and mutual exclusion.

The remaining three steps comprise the detailed design phase. During detailed design, each process identified in preliminary design is itself decomposed and described in sufficient detail for implementation. These processes can then be gathered together into modules exhibiting low coupling and high cohesion.

1. Design each sequential process. If necessary, decompose the process with DFDs. Use structured design to generate structure charts identifying the modules and interfaces in the process. A program design language (PDL) is used to describe the individual modules.

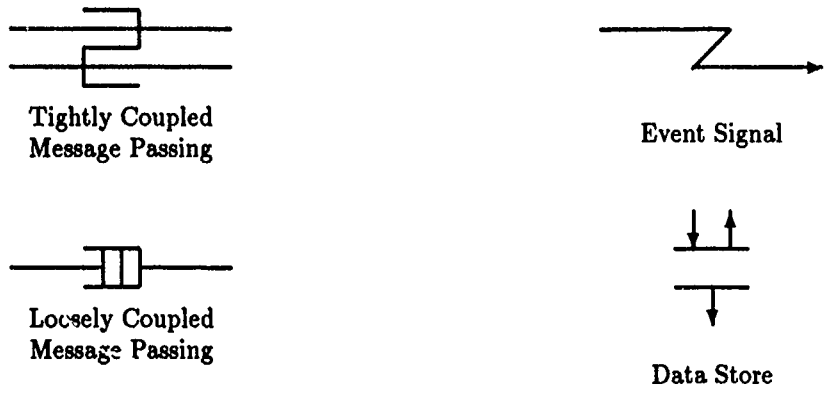


Figure 5.2. Process Interface Types

2. Design a state transition manager (STM) if necessary to maintain the current state of the system and a state transition table defining all legal and illegal transitions. A process calls the STM to determine if the desired action is legal for the current state.
3. Encapsulate the processes into modules (like packages for Ada). Objects used for intertask communications should be abstracted into data objects and encapsulated into modules.

5.2.2 Comments on Object Orientation Object Oriented Design (OOD) combines some of the features of structured and data structure design in an attempt to address the deficiencies of each. According to Booch(9), structured design focuses on functions or operations that must be performed with little consideration of the data structures on which they act. Data tends to be global, with the result that changes in data ripple through the entire structure. Data structure design concentrates on the data structures, treating the operations globally.

OOD, however, focuses on the definition of *objects,* each with its own set of applicable *operations.* Each module in the system denotes an object or class of objects. OOD places equal emphasis on the operations and objects on which they operate, saving the actual implementation until all objects, operations, dependencies, and interfaces have been adequately defined.

Object oriented design is not actually a software design methodology, "it is a *concept* that supports modern software design paradigms" (51:206). Application of object orientation encourages abstraction, information hiding, and modularity in the design process. Though object orientation is not an ARTMOS requirement, it is a natural offshoot of the requirements for modularity, portability, and verifiability. Indeed, object oriented design is a major component of LVM/OOD. OOD is emphasized where possible in the ARTMOS design, particularly during the detailed design phase.

5.3 ARTMOS Preliminary Design

This section outlines the preliminary design of the AFIT Real-Time Multiprocessor Operating System using the LVM/OOD-based methodology described in Section 5.2.1. The graphs and diagrams described by Nielsen and Shumate (51) are used to depict the stages of the design. Notational changes have been made where appropriate for the ARTMOS design and are explained.

5.3.1 Determination of Hardware Interfaces A context diagram for the ARTMOS is potentially very complicated. Because the ARTMOS will be running on multiple processor nodes, the number of hardware devices the ARTMOS could interface with is greatly increased. However, the requirements do not define a need for inter-processor interrupts or the capability for the ARTMOS running on one processor to directly affect the operation of another. Indeed, a basic assumption in the ARTMOS requirements analysis is that, except for certain distributed management functions which require information from other processors, all ARTMOS processing is limited to the local processor node. This simplifies the context diagram such that the ARTMOS only interacts with the local processor hardware.

The context diagram defines the hardware devices with which the ARTMOS interfaces and the data flow between the devices and the ARTMOS. An important distinction, discussed in Section 4.3.1, is that all ARTMOS hardware interaction is with the computer hardware, not the controlled system. For this reason, the context diagram need consider only those hardware devices *in the computer hardware* with which the ARTMOS interfaces.

Further complicating the definition of the ARTMOS hardware interfaces is that a primary objective of the ARTMOS is to establish an application program interface suitable for large real-time parallel systems with a variety of target parallel architectures. It is not practical to describe every possible device interface that could be on the different systems. (Attempts *are* being made to standardize the interface between hardware devices and real-time operating systems to simplify the design of device driver software for different hardware platforms (93). The draft interface standards are still a ways from universal acceptance, however.) Certain basic devices, however, are assumed by the ARTMOS and should be available on any system on which ARTMOS is implemented. These are: a timer device and communications hardware. An interrupt structure is also assumed, but this is the means by which devices "trigger" the operating system, not a specific device.

Obviously, a processor node has more devices than just a timer and communications hardware. All devices on a node must be initialized and have some device driver provided. These drivers, however, are not essential parts of the core ARTMOS functionality. Functions that utilize these hardware devices might be built above the ARTMOS, or be needed by the application, but are not part of the ARTMOS definition. Different applications can require vastly different capabilities in terms of hardware support. The pilot/vehicle interface in an advanced aircraft, for example, requires extensive and application-specific graphics and display equipment. Software to drive this equipment is part of the application functions, not the operating system. Other, less application-specific, devices may be needed simply as part of the computer hardware, such as a math coprocessor. The ARTMOS needs to provide additional handlers to initialize all such local and global devices at the node. The resulting context diagram is shown in Figure 5.3.

As shown in the context diagram, all devices are initialized by the ARTMOS. The ARTMOS has one data input, data messages received at the communications hardware, and one data output, data messages passed to the communications hardware for sending. Hardware control inputs, in the form of interrupts, are sent from the timer and communications hardware, while hardware commands are sent from the ARTMOS to the communications hardware.

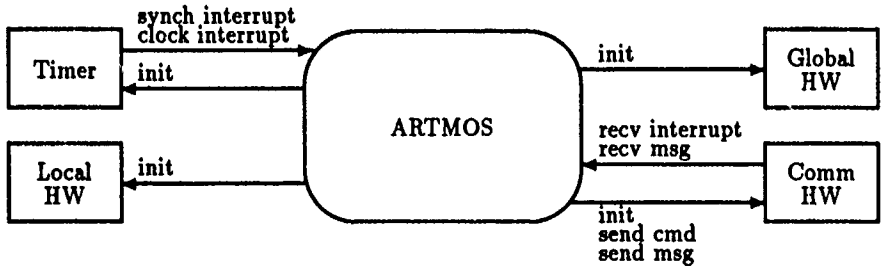


Figure 5.3. Context Diagram

5.3.2 Assignment of Processes to Edge Functions The second step of the methodology assigns control processes to each of the external devices, called "edge functions." These processes are simple device drivers and should contain as few instructions as possible. Also, the drivers should mask specific device-dependent details from the operating system. The product of this step is a preliminary concurrent process graph showing the device processes, with the remainder of the ARTMOS (the "middle part") shown as one abstract activity.

Rather than construct one preliminary process graph as described above, the decision was made to define two separate process graphs, one for ARTMOS initialization and one for the main execution section. Initialization is a one time activity, performed when the system powers up, to place system hardware devices and software data structures in a known state prior to operation. Once the initialization is complete, the execution section begins normal ARTMOS operation. The only link between the two activities is the start signal passed to the execution section when initialization is complete. Since these activities are essentially two independent operating modes, the decomposition is performed separately.

All hardware devices at the local processor must be initialized by the copy of the ARTMOS resident at the processor. Of the local devices, the context diagram explicitly

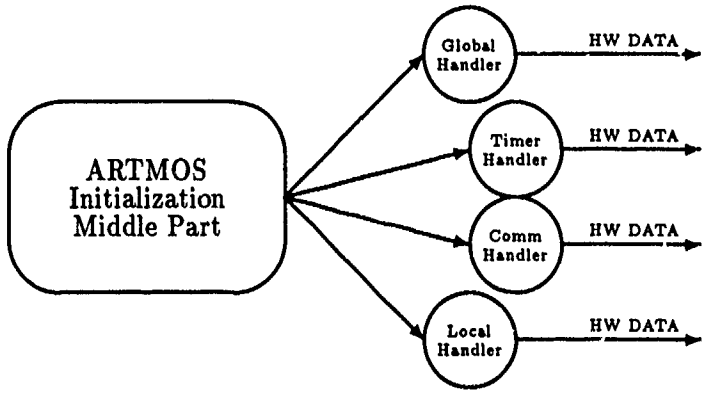


Figure 5.4. Preliminary Initialization Process Graph

identifies the timer and communications devices as needing individual attention by the ARTMOS. Separate processes are assigned to initialize the timer, communications, and any other local devices. Also, one of the processors is designated the master by the application. This processor is "responsible" for initializing any global hardware or mechanisms shared by the processors. An additional device process is needed to perform the initialization of the global hardware. The preliminary concurrent process graph for the initialization portion is shown in Figure 5.4.

In the execution portion, the ARTMOS interfaces with two devices: the timer and the communications hardware. Each device needs two device interfaces, however, so four processes are required. The timer interrupts the ARTMOS for two different events, time to increment the clock and time to synchronize the clock, each of which is handled differently. The communications hardware, on the other hand, has an input and an output interface. Figure 5.5 is the execution section preliminary process graph. The remaining functionality of the ARTMOS is contained in the middle part.

The clock interrupt handler clears the timer interrupt and performs any reset operations required by the timer, then notifies the ARTMOS to increment the local clock value. The synchronize clock interrupt handler also restarts the timer, after which it triggers the

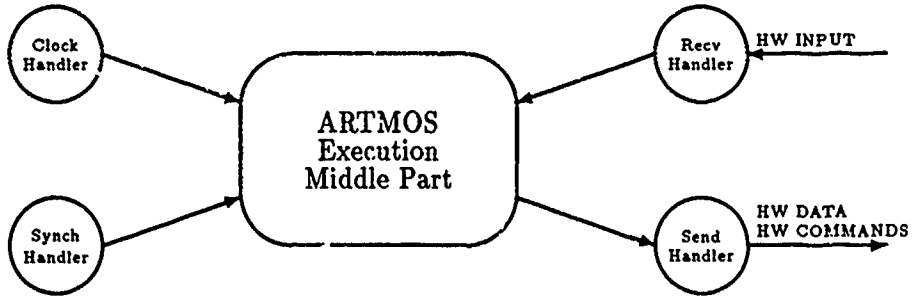


Figure 5.5. Preliminary Execution Process Graph

ARTMOS to adjust the local clock. Neither of these handlers actually perform ARTMOS clock operations; rather, they simply interface with the timer device and trigger ARTMOS to perform the indicated function. This approach allows different timer devices to be used without having to alter the ARTMOS clock increment or synchronization algorithms.

The receive message and send message handlers are more complex. The receive message handler accepts messages that arrive at the processor. The message arrives in some format and by some protocol determined by the specific communications hardware. The handler accomplishes the hardware communications protocol, converts the message from the hardware format to the operating system-defined format, and passes the message to ARTMOS for processing. The send message handler accepts a message from ARTMOS in the operating system format, converts it to the hardware format, and uses the hardware protocol to send the message. The key to this approach is that all details of the communications hardware are masked by the device handlers and thus are transparent to the ARTMOS.

5.3.3 Decomposition of the Middle Parts This step decomposes the middle parts of the preliminary process graphs to sufficient depth to identify all major functions and their

components. Both DARTS and LVM/OOD use data flow diagrams to analyze the data flow of the system and define the functions that must occur. According to Gomaa (25:948–949), there is no need to distinguish between control and data flow until the task interfaces are defined (in Section 5.3.5). Though not required by the methodologies, a different decomposition notation, such as SADT, that also depicts control flow and implementation mechanisms could also be used to decompose the middle parts (25:949). Since SADT was used in Chapter 4 to graphically decompose and specify the ARTMOS requirements, that notation is also used here.

As stated above, the middle part decomposition should continue until all major activities and their components have been identified. The purpose is to define *what* functions must be performed, not to describe *how* to implement these functions. Some of the decomposition performed in the specification is not included at this juncture because it pertains to the lower-level operation of the functions. As a result, the SADT diagrams produced here represent a *functional* restatement of the problem specification, including the edge functions.

Several small changes were made to the SADT notation for this decomposition. First, the processes identified in Section 5.3.2 for the external devices are represented as circles rather than as rectangular activities. This was done to emphasize the difference between the hardware drivers and the rest of the ARTMOS function. Second, signals to and from the hardware drivers that contain no data are represented by thin lines instead of the thicker lines used elsewhere. For example, the interrupt signals (which pass no data) from the Clock and Synch Handlers in Figure 5.7 are *thin* lines. The signal from Recv Handler, which passes the message, is a thick line. Finally, many of the hierarchical conventions of SADT are left out to allow the decompositions to fit on one page for clarity.

Figure 5.6 is the decomposition of the ARTMOS initialization section. This figure shows the major functional activities identified during the requirements specification and the control flow between them. Upon invocation of the ARTMOS, every processor initializes its local hardware devices and data structures. The ARTMOS Init Local Hardware activity interfaces with the device handlers for the local timer, communications hardware, and any other devices on the local node. Any interrupt handlers provided by the applica-

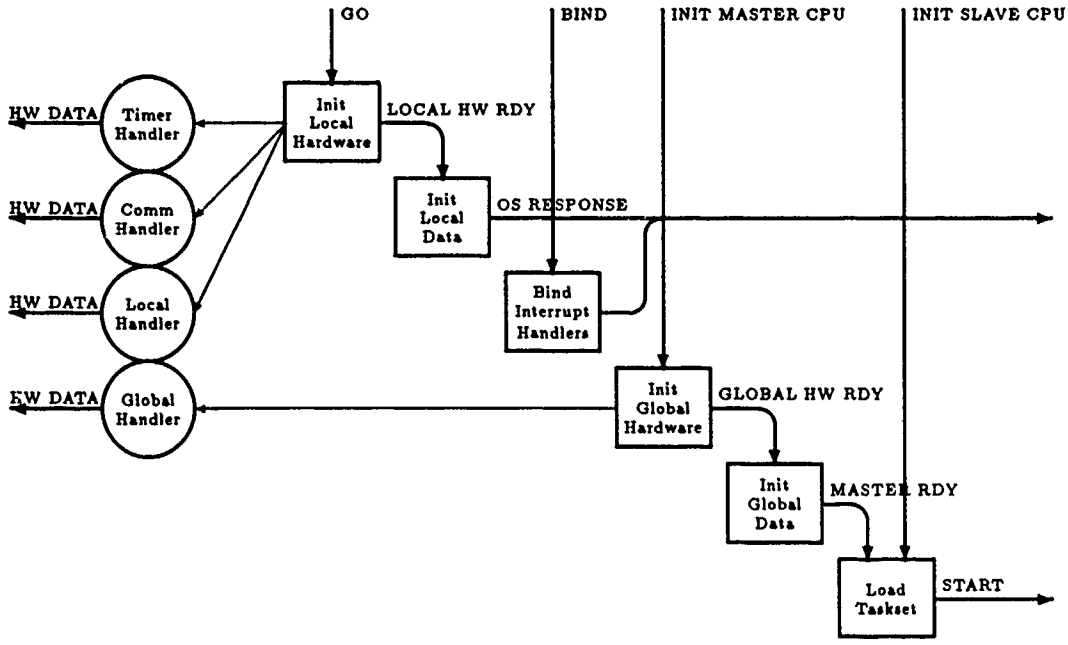


Figure 5.6. Initialization Decomposition

tion software are mapped to the appropriate hardware interrupt vectors. If the application software identifies this as the master processor, any global hardware devices and data structures are initialized, after which all the slave processors are signaled that the master is ready. The global hardware handler process performs the actual hardware initialization. Each slave processor then loads its taskset and passes control over to the execution section.

The decomposition of the initialization section is different from its original specification (as shown in Appendix A) due to insufficient analysis of the initialization requirements during the specification phase. When the specifications were reviewed in order to identify the major functional activities, several inconsistencies emerged. For example, the original specification calls for the application-specific initialization to occur before the local processor hardware is initialized -- which may prevent the application initialization from completing properly. This problem is corrected in the design.

The decomposition of the ARTMOS execution section is much more complex, as

shown in Figure 5.7. Each of the system call and interrupt handlers is drawn as a separate major function. The system calls could have been grouped as one activity, or shown as different classes of system calls, such as task or communications management, but these approaches would have been too high-level. Each of the calls performs a separate, significant function and should be portrayed as such. Each handler is triggered by the appropriate system call, performs the specified actions, then returns control to the application task or causes a new task to be scheduled. Likewise, the interrupt handlers are unique activities, each triggered by a different interrupt. Each interrupt handler has an associated device handler process. The device handler does all device-dependent processing, while the interrupt handler performs the device-independent actions.

The remaining two activities of the decomposition are Save Running Task and Schedule Task. When one of the system calls or interrupts has altered the list of ready tasks, Save Running Task is activated to check the list. If a ready task has a higher priority than the running task, the state of the running task is saved and it is put back in the ready list. Otherwise, control is returned to the running task. If the running task was saved, or if actions taken by the system calls or interrupts blocked or terminated the running task, Schedule Task selects the highest priority ready task, restores its state, and passes control to that task.

5.3.4 Identification of Concurrent Processes The forth design step is to use the ARTMOS functional decomposition from Section 5.3.3 and a set of process selection heuristics to combine activities in the middle part into concurrent processes. The goal is to abstract the SADT activities into independent processes that have a minimum of interprocess communications and data dependence. This section outlines process selection heuristics proposed by the authors of LVM/OOD (51:89–91), then applies these heuristics to the ARTMOS initialization and execution middle part decompositions.

5.3.4.1 Process Selection Heuristics The LVM/OOD and DARTS methodologies both present heuristics for the selection and abstraction of transforms or activities into concurrent processes. As the DARTS heuristics comprise a subset of the LVM/OOD list, only the LVM/OOD heuristics are described here.

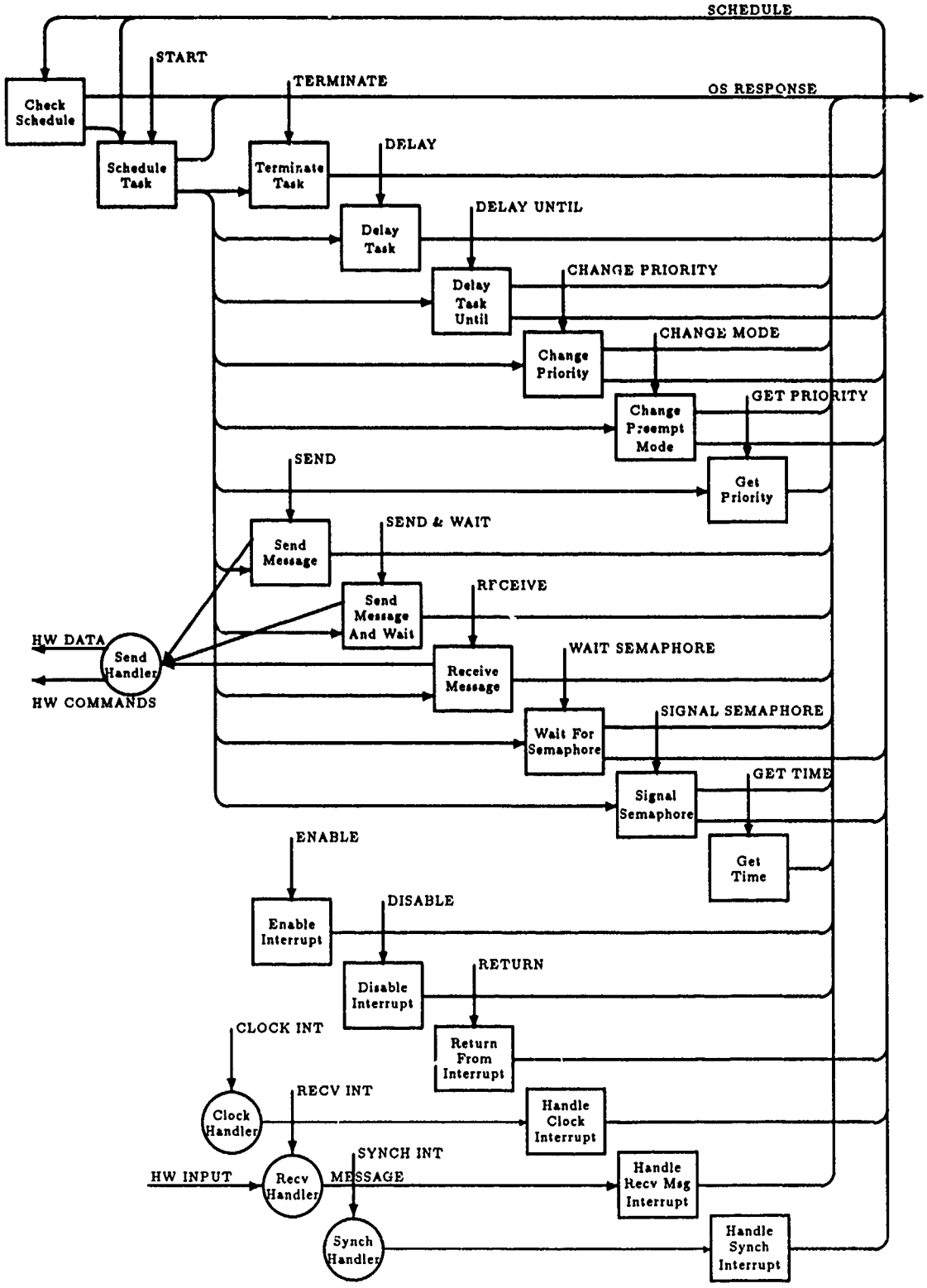


Figure 5.7. Execution Decomposition

1. *External devices.* Such devices typically run at different speeds and usually require a separate process for each device or channel. This criteria is the basis for step 2 of this design methodology (see Section 5.3.2).
2. *Functional cohesion.* Combine closely related functions into a single process if this reduces the overhead of managing each as a separate process.
3. *Time-critical functions.* Time-critical functions should be implemented as separate processes to ensure that real-time deadlines are met.
4. *Periodic functions.* Periodic functions should not be combined into a single process because of the overhead and difficulty with managing the different periods.
5. *Computational requirements.* Computationally intensive functions that are not time-critical can be designed as separate background processes.
6. *Temporal cohesion.* Functions that occur during the same time period or immediately following the same event should combined into a single process.
7. *Storage limitations.* - Hardware limitations such as the number of address bits may limit the number of processes.
8. *Data base functions.* - Functions accessing a shared data base can be collected into a single process with mutual exclusion of the access mechanism and the data structure hidden.

5.3.4.2 Selection of Initialization Processes For the initialization section, process selection is relatively straightforward. One factor simplifying the selection is the fact that initialization is not considered to be time-critical. Also, the external device handlers identified in Section 5.3.2 are separate processes, as specified by the first heuristic above.

All processors initialize the local processor resources when ARTMOS is invoked. This initialization consists of two separate activities, initialize local hardware and initialize local data structures, that are only performed once, sequentially. By the concept of temporal cohesion, these two activities are combined into one process, Init Local Resources.

Likewise, the global initialization is temporally cohesive and so should be one process. The master processor, as designated by the application software, is responsible for

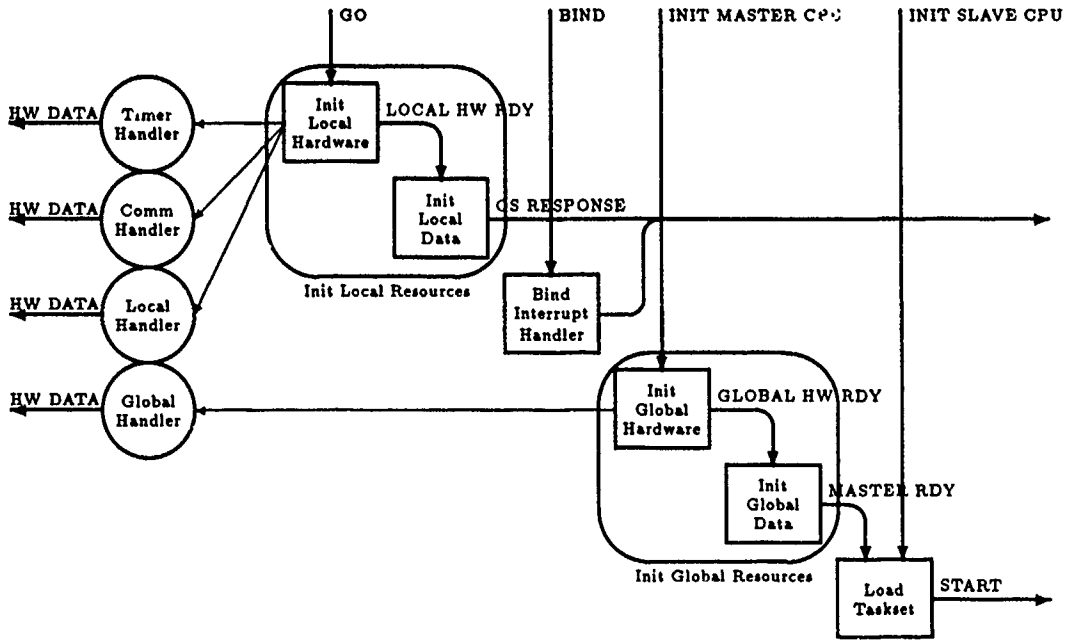


Figure 5.8. Initialization Decomposition and Process Identification

the initialization of all global resources. This consists of initializing all global hardware resources and software data structures. These two activities always occur together after the application software requests the ARTMOS to perform master processor initialization.

The binding of interrupt handlers on each processor is independent of the other initialization activities, and is separately activated by the applications software. It should be a separate process. The final function is Load Taskset, which is performed on all processors when the master signals it is ready. It is an independent activity, and should be a separate process.

Figure 5.8 is the process identification for the initialization section. Rather than using shaded boxes around the functions to indicate process selection, as done in LVM/OOD and DARTS, the rectangular functions were enclosed in ovals. This change was made because of the density of the execution section process identification developed in the next subsection.

5.3.4.3 Selection of Execution Processes The ARTMOS execution section is much more complicated than the initialization section described in the previous subsection. As before, the device drivers are all separate processes.

With the exception of the Save Running Task and Schedule Task activities (and the four device driver processes), all of the activities in the execution functional decomposition (Figure 5.7) are independent. Each is triggered by some unique external signal (such as a system call or interrupt), perhaps has some input, and then returns control to the application or triggers rescheduling operations. All activities occur at intervals completely controlled by entities outside the operating system (such as the hardware or application software).

In a sense, all ARTMOS execution functions are time-critical. Every moment spent in the operating system could delay an interrupt, or cause an application task to miss a deadline. While the ARTMOS does not itself operate with specific deadlines, it must minimize the time spent out of the application software. This is a strong argument for separating all of the functions into separate processes.

By analogy, the system call handlers can be viewed as "interrupt handlers" for software "devices." The primary difference between the hardware and software "devices" is that a unique handler is associated with each hardware device, while one system call handler processes requests from any software task. The call handlers are essentially server processes, receiving requests to perform a specific action for some task. Because of the asynchronous nature of the call and interrupt handlers, their time-criticality, and the similarity of system calls to interrupts, each system call and interrupt handler should be a separate process.

Should Save Running Task and Schedule Task be one process or two? The operations performed by these functions are related, and tend to occur sequentially, though sometimes only one of the two takes place. If a system call requests some action that blocks or removes the calling task (such as the Terminate Task call), the call handler triggers Schedule Task to schedule a new task for execution. If, on the other hand, a call or interrupt handler changes the ready task list, a new task *may* need to be scheduled. Save Running Task is triggered

to check the ready task list and save the running task's state if a higher priority task is ready. If no higher priority task is ready, control is returned to the calling or interrupted task. Otherwise, Schedule Task is triggered. Because of the unusual interaction between the activities, they are not consistently temporally cohesive. They should be separate processes.

Therefore, every activity identified in the execution section decomposition is a unique process. Figure 5.9 shows the process identification. Note that, as mentioned in the previous subsection, processes are represented by ovals rather than shaded boxes around the activities. This decision was forced by the complexity of the diagram, in an attempt to keep it readable.

5.3.5 Determination of Process Interfaces In the previous four steps, the ARTMOS hardware interfaces have been identified and the middle portions of both the initialization and execution sections have been decomposed. Concurrent processes have been identified for each section. These processes may access common variables, or respond to signals or parameters received from other processes. In the final step of preliminary design, the interaction between the processes is formally defined. The interfaces between communicating processes is determined by considering the desired coupling between them. Process structure charts are used to shown the processes and interfaces for both the initialization and execution sections.

This section describes the interfaces defined for the ARTMOS processes. The first subsection considers the need for shared variables in the ARTMOS. The following two subsections discuss the interface definition and process charts for the ARTMOS initialization and execution portions.

5.3.5.1 ARTMOS Shared Variables One possible problem with the ARTMOS design as it has been described so far is that several data structures are needed by most of the processes. In particular, the pointer to the currently executing process (Running Task) and the task management lists (Ready Tasks, Sleeping Tasks, and Blocked Tasks) all are heavily used. Running Task is read by all but four of the system call activities, though

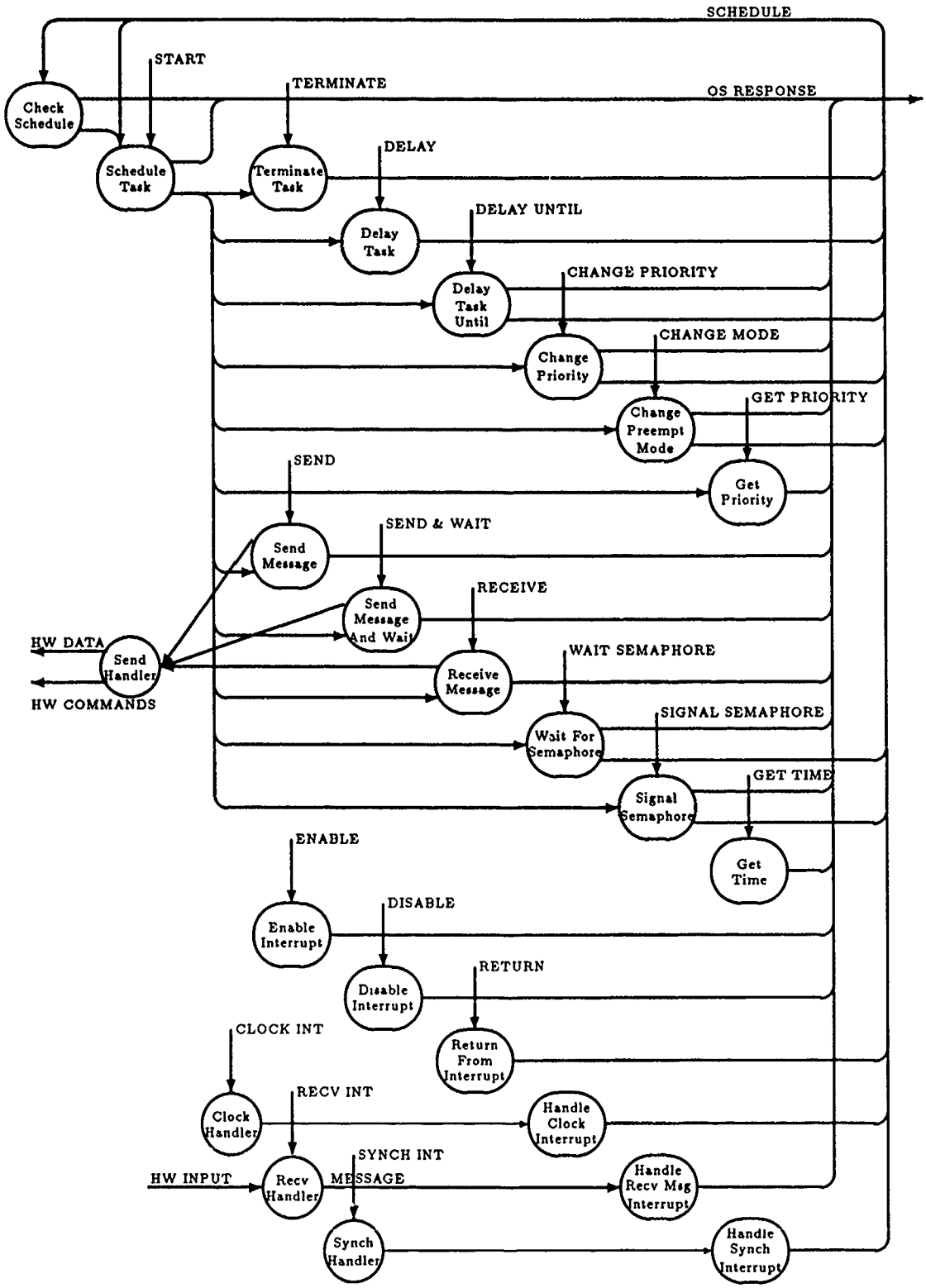


Figure 5.9. Execution Decomposition and Process Identification

only the Schedule Task process can write to the variable, as shown in the specification document and process decompositions. The task management lists are read and written by many of the system call and interrupt handler processes.

The problem with these data structures is that the processes (system call and interrupt handlers) that use them are invoked asynchronously. The handlers are not called in the same way that procedures are; rather, they are activated by external events, either system calls or interrupts, that occur at unpredictable intervals. Many of these handler processes perform some service for the calling task or change the set of executing tasks. In order to manipulate the calling task or task sets, the handlers have to be able to access the pointer to the running task or the task management lists. The difficulty stems from the asynchronous nature of the processes. Since they are not "called" in the conventional sense, operating system data variables like Running Task and the task lists cannot be passed to the processes as "parameters." The only information passed to the system call handlers is parameters to the calls, supplied by the application. Since Running Task and the task lists are hidden from the application, it cannot pass them to the system call handlers. The only data available to the handlers, then, is either passed from the application, encapsulated in the handlers, or globally accessible operating system data.

Given the necessity of shared variables, the question arises as to whether some form of monitor processes are needed to protect access to the shared variables. Nielsen and Schumate introduce intermediary processes, called monitors, which encapsulate the shared data items and their associated access mechanisms. Processes needing to access the shared data use the provided mechanisms, thus guaranteeing the integrity of the data. For the specific case of the ARTMOS task variables, such monitor processes are not appropriate for the level of detail. Manipulation of the running task and the management lists occurs at a very low-level, internal to the major functional components of the ARTMOS considered in the preliminary design. Task management processes, for example, do not directly manipulate *tasks*, but rather they operate on Task Control Blocks (TCBs), which are data structures used by the ARTMOS to store the context of the tasks. It is these low-level operations that must access the global variables. (TCBs are an integral part of the detailed design.)

Monitor processes may not be necessary anyway. Running Task, for example, is used by many of the system call and interrupt handlers, but only written by one process: Schedule Task. Since the handlers and Schedule Task occur sequentially rather than concurrently, there are no conflicts over the variable. The task lists are read and written in many processes, but the variables are safe in this case as well. Only one system call can be invoked by the application at any given time, access to the variables is non-overlapping. The potential problem stems from the interrupt handlers. If an interrupt occurs during the modification of a task list, the interrupt may read invalid data. All ARTMOS activities should mask interrupts during access to the shared variables to prevent this situation.

The specific structure of the globally accessible variables and the routines for accessing the variables is discussed at greater depth in the detailed design.

5.3.5.2 Initialization Process Interfaces The initialization processes identified in Figure 5.8 are pictured in the process structure chart of Figure 5.10 which defines all interfaces between the processes. With one exception, all initialization communication consists of event signals passed between processes. These events do not pass data, they simply signal some event for which the destination process is waiting. For example, process Init Local Resources is triggered by an event, GO, which signifies that the operating system should begin execution. No parameters are needed. Processes Init Global Resources and Init Local Resources do not pass data to the device handlers, they simply signal the handlers to begin operation. When initialization is complete (the local taskset has been selected and loaded), Load Taskset triggers the start of the ARTMOS execution portion. The coupling between these processes is neither loose (a message queue) nor tight (a message/reply).

The event signal between Init Global Resources and Load Taskset is unique in that it occurs between processors rather than on just one processor. When the master processor is ready, it signals the Load Taskset process on all the processors to begin. This synchronizes the processors initially.

The remaining process, Bind Interrupt Handler, is more tightly coupled to the application than the other processes because it receives parameter data. This data consists

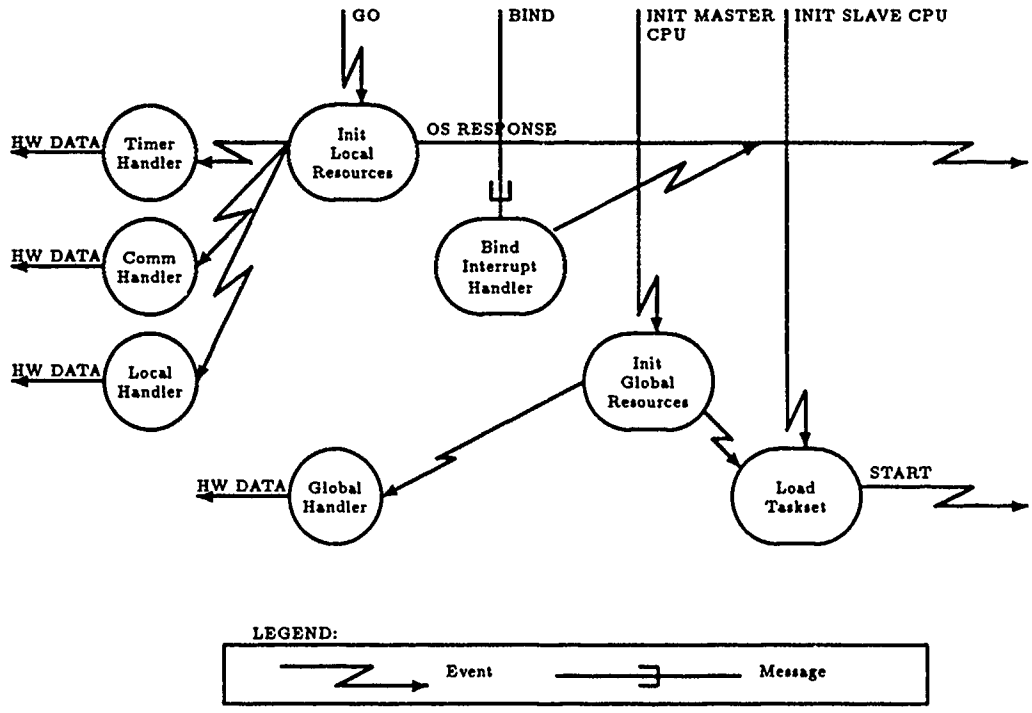


Figure 5.10. Initialization Concurrent Process Graph

of some interrupt information needed by the process to bind the application-provided interrupt handler to the specified interrupt. As a minimum, this data must consist of an interrupt identifier and a pointer to the handler.

5.3.5.3 Execution Process Interfaces Figure 5.11 shows the processes identified in Section 5.3.4, along with the interfaces between the processes and the application. There are essentially three types of communication that occur between these processes. First of all, the system call handler processes are all activated by a system call from the application. Some of the calls are parameterless, such as Get Priority or Terminate Task, because they do not need information from the calling task to execute. These parameterless calls are represented by event signals, as in the initialization section. Other system calls require the application task to provide some information. These processes are shown as receiving a *message* from the application rather than an event. For example, process Delay Task needs to know how long the calling task wishes to be delayed, just as Send

Message needs to be given a message to send. As expected, the interrupt handler processes are activated by the interrupt event; they are passed no data.

The second type of process interaction occurs between the "middle part" processes and the device handler processes. Clock Handler and Synch Handler, for example, simply signal the interrupt to the Handle Clock Interrupt and Handle Synch Interrupt, respectively. Recv Handler passes a message in operating system format to process Handle Recv Msg Interrupt. A loose coupling is required between these processes since the messages arrive asynchronously. This allows Recv Handler to queue up messages as they arrive, to be handled by the Handle Recv Msg Interrupt at its own pace. The Send Handler process is passed a message to send to a mailbox. Messages can originate at any of the communications management activities, Send Message, Send Message and Wait, and Receive Message. Though messages are passed to Send Handler at a rate determined by the application, Send Handler is forced to operate at the rate of the communications device. The processes must therefore be loosely coupled to allow buffering of the messages.

The final type of process interaction is between the scheduler processes, Save Running Task and Schedule Task, and the rest of the processes. All of the handler processes, upon comlletion, either pass control back to the invoking application task (with or without response data) or request a scheduling action. In the latter case, an event is triggered to either perform a rescheduling operation if needed (if the set of ready tasks has changed) or to explicitly perform rescheduling (if the running task has been blocked, terminated, or delayed). No parameters are needed to request rescheduling, so events are used. Once rescheduling has occurred, control is passed to the new running task. Process Schedule Task updates the global Running Task variable.

5.4 Nature of ARTMOS Processes

From the identification of execution processes, it was decided that all of the major activities should be separate processes. This is reasonable, as the ARTMOS is really just a collection of independent, asynchronous functions. Interaction between the system call and interrupt handler processes is non-existent. The question that arises, then, is whether these functions should actually be concurrent *processes* or "merely" a set of specialized

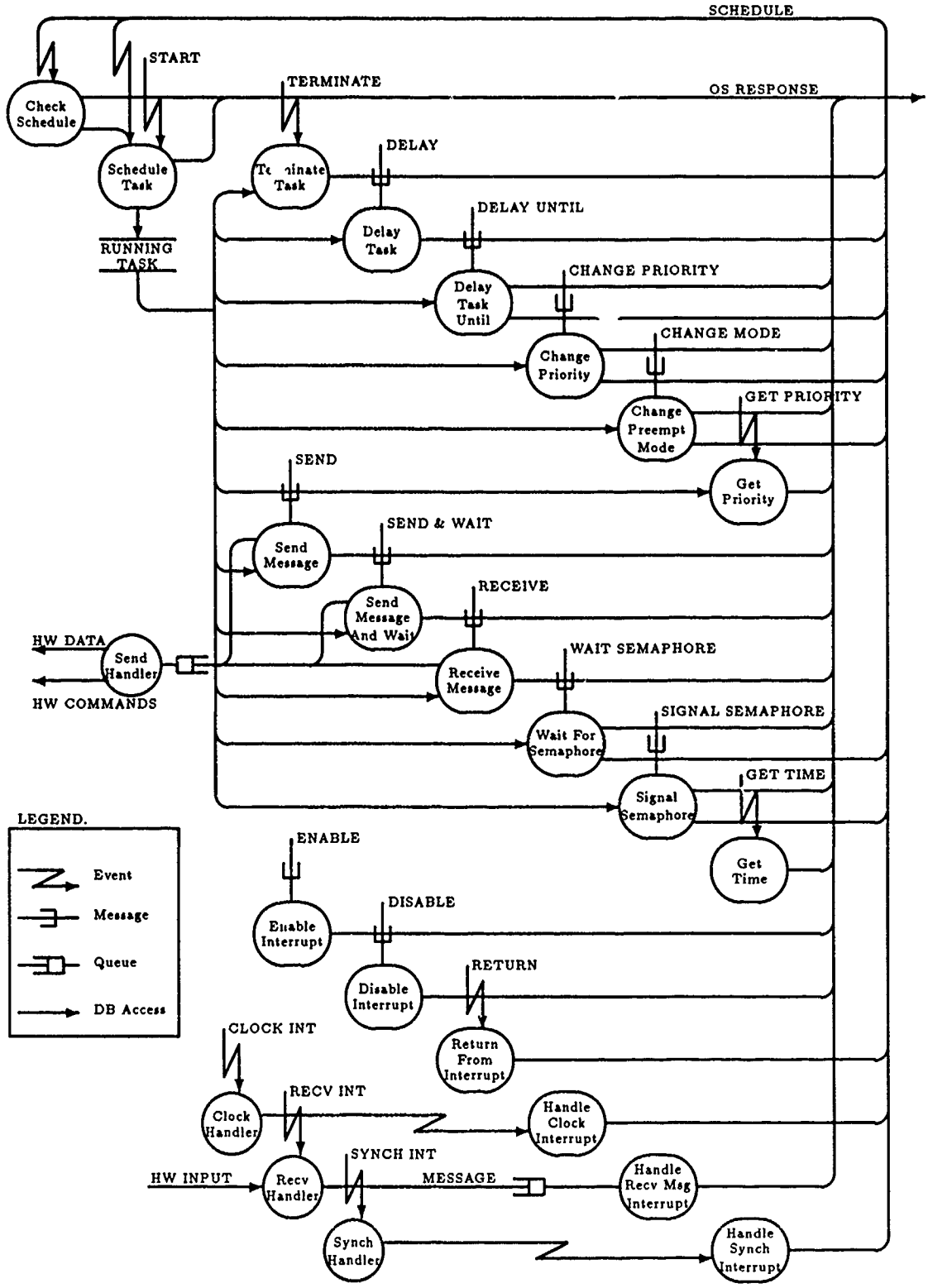


Figure 5.11. Execution Section Concurrent Process Graph

handler *subprograms*. What are the benefits or disadvantages of either approach? Further, is a hybrid of these two approaches possible or desirable?

While concurrent processes excellently model the asynchronous and real-time nature of the problem domain, a certain degree of overhead is necessary to schedule and control the process. Interestingly, the major function of the ARTMOS is to provide these process control primitives to the application software. There is no reason why these same facilities cannot be used by the operating system itself. Several problems arise from this approach, however. For the ARTMOS initialization section, processes are not possible because the process control mechanisms are not available until the execution section begins. The device initialization handlers only execute once, so do not have to operate asynchronously to match the speed of the hardware devices. During the execution section, use of processes is not appropriate for many of the system call activities. For example, it seems excessive to go through the overhead of activating and scheduling an operating system task whose function is to terminate the calling application task. Likewise, using a task to schedule tasks makes little sense. Further, it is not possible for two system calls to overlap, since the operating system does not release control to the application until it has completed the first call.

In some cases, it might be useful or desirable for a call or interrupt handler to spawn a process to handle some non-critical or computationally intensive activity. For example, the clock synchronization handler could activate a task to communicate with similar tasks on other processors to determine the new time. For the moment, all of the handlers are viewed as subprograms. If the detailed design of these handlers identifies the need to use processes, they are incorporated at that point.

With the decision to go with subprograms rather than processes, the importance of the selected design methodology is not reduced. First of all, LVM/OOD explicitly considers hardware interaction, something that is not possible with structured design. More importantly, structured design is a means of modeling a hierarchical decomposition of functions. It is not intended to model independent, potentially concurrent operations the way DARTS and LVM/OOD are. The ARTMOS, consisting of many independent, asynchronous activities, does not lend itself well to hierarchical design. The selected design

methodology enabled the development of a flexible, consistent framework in which to design the ARTMOS functions.

5.5 Chapter Summary

This chapter outlines the preliminary design of the ARTMOS. Several design methodologies suitable for real-time software are discussed, and a new methodology combining some of the attributes of them all is presented in detail. Specific hardware devices interfacing to the ARTMOS are identified, and independent processes are allocated to handle the devices. From there, the decomposition of the ARTMOS software system is partitioned into initialization and execution sections. This partitioning is based on the realization that initialization is a one-time, non-time-critical activity, and therefore has different characteristics than the "normal" operation of the ARTMOS. For each section, the "middle part" is functionally decomposed to identify the major activities that must be performed. Using process selection heuristics provided by LVM/OOD and DARTS, the functions are grouped together or isolated into concurrent, communicating processes. The process interfaces are then defined according to the desired coupling between the processes.

The next chapter presents the decomposition of the concurrent activities identified during preliminary design into the final detailed design suitable for implementation.

VI. Detailed Design

This chapter describes the detailed design phase of the ARTMOS development. The preliminary design developed in Chapter 5 presents a high-level view of the ARTMOS structure and major functional activities. This view defines the major functional components of the ARTMOS and the manner in which the components interact, but does not consider how each component performs its function. Preliminary design essentially defines a *framework* for meeting the functional requirements of the system defined in the functional requirements specification (Chapter 4). Detailed design, as the name implies, provides the low-level details necessary to implement the design. The functional activities defined in the preliminary design are decomposed until they are of sufficient detail that they may be relatively easily translated into a selected programming language. Specific algorithms and data structures are designed.

An additional design consideration is the selection of a target computer system for the ARTMOS. To this point, the design has been independent of any specific parallel architecture. The detailed design phase binds the overall system design to a particular target architecture (58:130).

This chapter provides an in-depth description of the detailed ARTMOS design. It first briefly outlines the software engineering representation notations that are used in this stage of the ARTMOS development, and how they are applied. The next section describes the selection of a specific target hardware architecture to be used for the laboratory implementation of the ARTMOS. It considers the manner in which the "virtual shared memory" programming model is implemented with the target architecture. The final three steps of the design methodology developed last chapter are described in detail. This discussion includes various alternatives considered and decisions made during the detailed design process.

6.1 Design Methods

Both DARTS and LVM/OOD recommend the use of structured design to decompose each of the independent processes identified during preliminary design (Section 5.3.4).

Structured design uses data flow diagrams to model the flow and transformation of data in the processes, then applies transformation and/or transaction analysis to convert the data flow diagrams into structure chart notation.

The decision was made early in the ARTMOS development to utilize SADT diagrams in place of data flow diagrams because of the explicit depiction of control flow allowed with SADT. The requirements specification document uses SADT diagrams to formally specify the ARTMOS requirements. Rather than convert the existing SADT diagrams to data flow diagrams to decompose the processes using structured design, the transform analysis is applied to the SADT diagrams. In fact, SADT can be used in place of data flow diagrams in the DARTS methodology, though DARTS does not distinguish between data and control flow until the process interfaces are defined (25:948–949).

Two extensions to structure chart notation are needed for the ARTMOS detailed design. First, as discussed in Section 5.3.5.1, ARTMOS uses some global variables, such as a pointer to the currently running task. In standard structure chart notation, an arrow with a black circle at the base represents a control flag passed between two subprograms. An arrow with a white circle shows data flow. A new symbol is added to represent access to global memory variables: an arrow with no circle. When this symbol is used, the pointed-to module uses the global variable; it is not passed as a parameter. When global variables are actually passed to a subprogram, standard circle notation is used.

The second extension is necessary because structure charts provide no means for showing the interaction between lower-level subprograms in the decomposition of one process and other, independent processes (44:74). A new notation is needed to represent independent processes as well as subprograms in the structure chart. As with standard structure charts, a rectangle represents a module or subprogram. An oval is added to show a call to an independent process. (44:74) also notes that each level of the structure chart decomposition does not now strictly imply a subordination of functions, as it would without the addition of independent processes.

While structure charts are an excellent tools for describing the component modules of an activity and the interaction between the modules, the internal design and control

flow of the modules is not apparent. An additional tool is needed to provide access to this lower-level information. Another purpose of such a tool is to help transform the design in graphical notation closer to the final code implementation (51:63). Graphical and table-driven representations, such as flowcharts, box diagrams, and decision tables (58:245–252), can show the internal decisions of a module, but bring the design no closer to implementation. A program design language (PDL), on the other hand, combines the vocabulary of one language (e.g., English) and the syntax and structure of a programming language (58:253). A PDL "is used to translate the design in graphical terms into programming constructs" (51:63).

Nielsen and Schumate note that the PDL should be tailored to a specific implementation language when applied to a design (51:63). They recommend the use of an Ada PDL in their LVM/OOD methodology, which is itself based on the Ada language. Since the ARTMOS is intended to be language independent, however, a language-specific PDL may hinder more than help. For this reason, a generic PDL is used that provides the basic programming constructs needed to adequately depict the ARTMOS design.

The basic programming constructs used are:

1) IF < *condition*1 > THEN
   one or more statements
   ELSE IF < *condition*2 > THEN
   one or more statements
   ELSE
   one or more statements
   ENDIF
2) WHILE < *condition* > DO
   one or more statements
   ENDDO
3) REPEAT
   one or more statements
   UNTIL < *condition* >

These constructs provide a structured means of describing the important actions and decisions within a module while also moving the design closer to the actual implementation.

The PDL is used in conjunction with the structure charts in Appendix B to show the internal design of the modules, and also with the Data Dictionaries in Appendix C to describe the algorithms of each program module.

6.2 Selection of Target Hardware

The specific objectives of this thesis are to specify and design a modular real-time multiprocessor implementing the AMCAD application program interface, and map that design to a suitable target multiprocessor architecture available in the AFIT environment. The ARTMOS preliminary design phase described in Chapter 5 accomplishes the first objective: an architecture-independent specification and high-level design is complete. This chapter performs the second objective, that of mapping the preliminary design to a selected target machine.

The available parallel systems at AFIT are the Intel iPSC/1 and iPSC/2 message-passing hypercube machines and the Encore Multimax, a shared memory architecture. The decision was made to limit the candidates to these "general-purpose" parallel architectures rather than one more like the embedded multiprocessor systems targetted by the ARTMOS design. This decision is motivated by two concerns. First, the Encore or the hypercubes can serve as both development system and target system. An embedded target, such as a VMEbus rack or an aircraft flight control computer, would require an additional computer to be used as the development system, with all code downloaded to the target for execution and testing. Secondly, the goal of this thesis project is not to produce a commercial product. The "product" of this work is the specification and detailed design of a *prototype* operating system that is suitable for real-time parallel systems and can be used as testbed for real-time techniques and policies. An embedded target lacks the required development capability and flexibility.

Of the available systems, the iPSC/2 is selected for the ARTMOS detailed design. First of all, the iPSC/2 cube manager, called the Systems Resource Manager (SRM), has extensive tool support for the development, compilation, and debugging of the operating system and applications software, and facilities for loading the software onto the hypercube nodes. The SRM is directly connected to each node in the cube. The nodes, once loaded

with the appropriate software by the SRM, execute independently of one another and the SRM unless the software calls for communications or interaction. With this environment, software can be developed on the SRM and loaded onto the nodes for testing. The SRM in turn can act as a performacne monitor, collecting performance and status information from the nodes during ARTMOS execution.

Finally, the ARTMOS application program interface is essentially an extended shared memory model. Implementing a shared memory programming model on a shared memory machine such as the Encore proves little. By designing the ARTMOS for a loosely coupled, distributed-type architecture like the hypercube, the programming model is more fully exercised. While a hypercube architecture per se is not likely to be found in an aircraft system, it provides a much more realistic execution environment than would a shared memory architecture.

The remainder of this section is a brief overview of the iPSC/2 hypercube architecture. Interested readers can find more complete descriptions of the hypercube architecture (18:119–126), the iPSC/2 hardware and software architecture (18:441–456) and (38), and the iPSC/2 operating system (37).

A hypercube is "a loosely coupled multiprocessor composed of $N = 2^n$ processors interconnected as an n-dimensional binary cube" (18:119). Each processor node is a self contained computer with its own CPU, local memory, and communications facilities. Each node P_i is directly connected to n neighbor processors along the edges of the n-cube, as shown in Figure 6.1. Each processor's address differs from that of its n-neighbors by exactly one bit position. The hypercube has a maximum internode distance of $\ln N$ "hops", so any two nodes can communicate fairly rapidly.

As mentioned above and shown in Figure 6.2, each iPSC/2 processor node is an independent, funtionally complete computer. The node processor is an Intel 80386 microprocessor running at 16 MHz. Scalar processing support is provided by either an Intel 80387 or Weitek 1167 floating point coprocessor residing on the node, while an additional vector processing board can be attached to the node via the Standard Bus Interface. Each node can have up to eight MBytes of local memory on the node, while an additional 8

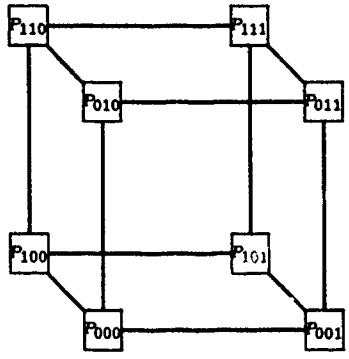


Figure 6.1. Hypercube Topology for $n = 3$ (18:122)

MBytes can be installed in the adjacent slot in the cube unit. In addition, each node has a 64 Kbytes data and code cache.

The iPSC/2 uses a Direct Connect routing module (DCM) on each node for communications. The DCMs link the the node backplanes as a circuit switched network, as well as providing an interface between the nodes and the network. The DCM provide for all hardware routing, unlimited message length, concurrent message paths within the router, and bi-directional message traffic, with a bandwidth of 2.8 MBytes/sec (18:441). These are all significant improvements over the iPSC/1.

The DCM provides two other features that substantially reduce the overhead of communicating between distant nodes. First, the iPSC/1 uses a "store and forward" protocol, where each node between the source and destination nodes is interrupted to determine whether the meesage should be processed or sent on. The iPSC/2 DCM hardware does not interrupt the node processor unless the message is intended for that node. Secondly, since the DCM establishs a circuit switched path between the source and destination nodes, a message sent between the two most distant nodes takes only slightly longer tha- ' etween two physically connected nodes. Message passing to be viewed as if the nodes are *fully connected*, with dedicated links between each two nodes, rather than implemented as a hypercube (38).

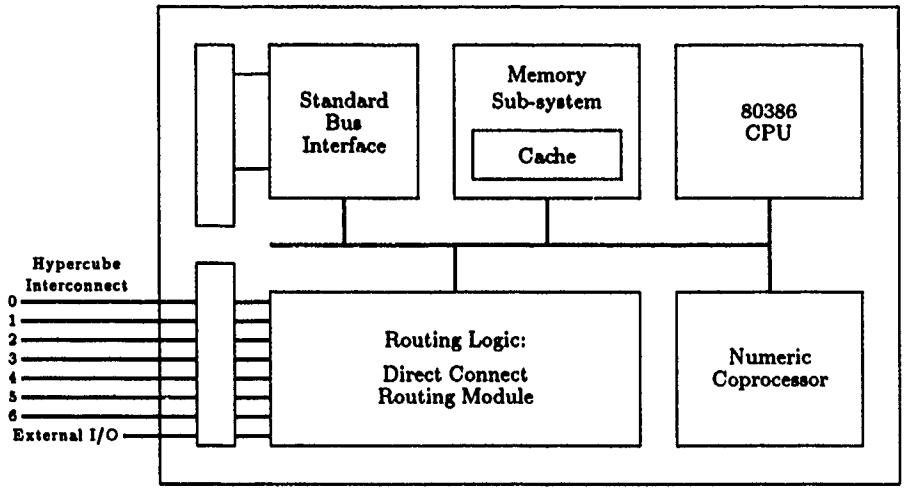


Figure 6.2. iPSC/2 Node Block Diagram (18:443)

6.3 ARTMOS Detailed Design

The ARTMOS design methodology, described in Section 5.2, describes three steps which must be performed in the detailed design stage. First of all, each of the identified concurrent processes is designed, using data flow diagrams if necessary to decompose the processes. Structure charts are used to identify the subprograms and their interfaces in each process, with pseudocode used to describe each subprogram. Secondly, a state transition manager must be defined if necessary to maintain the current state of the system and to validate state-dependent actions. Finally, the design is encapsulated into file modules, much as Ada encapsulates software entities into *packages*. These three steps are described in the following sections.

6.3.1 Process Design Section 5.3.4 identifies 8 concurrent processes for ARTMOS initialization and 24 processes comprising the main execution portion of the ARTMOS. During the detailed design, an additional initialization process was identified and is described below. This section describes the internal design of each of the processes. Struc-

tured design is applied to develop a structure chart decomposition of the processes (25:942). The actual design can be found in Appendix B, with Data Dictionary entries for the processes and parameters located in Appendices C and D, respectively.

Since the preliminary design decomposed the operating system into a collection of independent cooperating processes rather than a hierarchy of modules, some numbering scheme is needed to identify each process for further decomposition. The preliminary design partitioned the design into initialization and execution sections, so that method is used for numbering the processes as well. All initialization processes are numbered as $1.x$, where x can be 1 through 9. Execution section processes are numbered from 2.1 to 2.24.

6.3.1.1 Initialization The ARTMOS initialization function consists of three distinct phases, based on the need to synchronize the multiple processor clocks before beginning multitasking. These phases are: local resource initialization, application initialization, and system initialization.

6.3.1.1.1 Local Resource Initialization When each processor powers up, it begins local resource initialization to place all local operating system data structures and hardware devices in a known state prior to application execution. This process, INITIALIZE LOCAL RESOURCES, also executes upon system reset or ARTMOS restart. INITIALIZE LOCAL RESOURCES masks interrupts to prevent system interrupts from occurring before the operating system is ready to handle them, then calls INITIALIZE LOCAL HARDWARE and INITIALIZE LOCAL DATA STRUCTURES.

INITIALIZE LOCAL HARDWARE initializes all hardware devices on the local processor. Separate processes are defined which interface with the specific devices to reduce the reliance of the main ARTMOS code on particular hardware devices. One such process, TIMER HANDLER, programs the hardware timer on the local processor for periodic clock and clock synchronization interrupts. The specific periods for these interrupts will be determined during the coding phase, as appropriate for the specific class of application. A second process, COMM HANDLER, performs any initialization required by the local communications hardware, including the Direct Connect Module (DCM) and the Direct

Memory Access (DMA) controller (37:4.1,4.9–4.11). These two processes directly address those hardware devices used by the operating system: the timer and the communications hardware. Any other hardware devices on the processor are initialized by the LOCAL HANDLER routine.

INITIALIZE LOCAL DATA STRUCTURES creates and initializes all local processor data structures used by the operating system. The major data structures to be initialized include: the interrupt table, the memory list, the semaphore list, the communications queues (free receive queue, free send queue, and send queue), and the task management queues (free tcb queue, ready queue, and sleep queue). The format and initialization requirements of these data structures is described in the decomposition of the ARTMOS execution section processes.

6.3.1.1.2 Application Initialization Once the local processor resources are initialized, control is passed to the application software for initialization of its data structures and resources. Two ARTMOS system calls permit application declaration of interrupt handlers or semaphores. The application uses the BIND INTERRUPT HANDLER call to identify an interrupt service routine to be called when the indicated interrupt occurs. The handler maps the interrupt handler to a specific hardware interrupt vector on the local processor and sets up the appropriate interrupt table entry as indicated by the system call parameters. The application declares semaphores to the operating system with the BIND SEMAPHORE call. This system call adds the passed semaphore pointer to the local semaphore list. This list is used by the ARTMOS to check for tasks blocked in semaphore queues whose timeout has expired.

6.3.1.1.3 System Initialization The two previous initialization activities occur independently on all processors. The final initialization phase, system initialization, could also be called master/slave initialization. During master/slave initialization, one processor is selected as the "master" processor for system-wide initialization; all other processors act as "slaves" to the master processor. The master processor also performs the slave initializations, while commanding the others, so that the same initializations occur on each processor.

When the application completes its initialization, master/slave initialization is initiated by invoking the INITIALIZE SLAVE CPU on all the slave processors and the INITIALIZE MASTER CPU system call, followed by the INITIALIZE SLAVE CPU system call, on the master processor.

When the application issues the INITIALIZE MASTER CPU system call on the master processor, the INITIALIZE GLOBAL RESOURCES routine is invoked. This routine first initializes any hardware shared by the processors in the system, such as a communications network. It then initializes any data structures needed to manage the entire system. Both of these activities are shown in the ARTMOS design for flexibility and expandability, but neither needs to be performed in the ARTMOS iPSC/2 implementation. In the iPSC/2 there are no shared hardware devices other than the communications network, which is initialized automatically by the Direct-Connect Modules on each processor node. Likewise, since each processor manages a portion of the total system workload independently of the others, no shared data structures are required.

The final action taken by INITIALIZE GLOBAL RESOURCES is to signal the slave processors that the master is ready; that is, global initialization is complete. The slaves are signalled by the master processor sending a command message to each processor using the ARTMOS communications routines. The master initialization is now complete.

Slave initialization is performed by the LOAD TASKSET routine running on each processor and invoked by the INITIALIZE SLAVE CPU call. The routine busy waits until receiving the signal from the master processor that global system initialization is complete. When the processors receive the master ready signal, they each select and load the appropriate set of application tasks before calling SCHEDULE TASK to begin multitasking of the application tasks.

As discussed in Section 4.3.4, the ARTMOS tasking model calls for the partitioning of the application taskload into tasksets, one per processor. A taskset is statically assigned to each processor, based on the ID of the processor. Once a processor has loaded its static taskset, the processor manages the dynamic sheduling of the tasks according to some scheduling algorithm, described in Section 6.3.1.2.3.

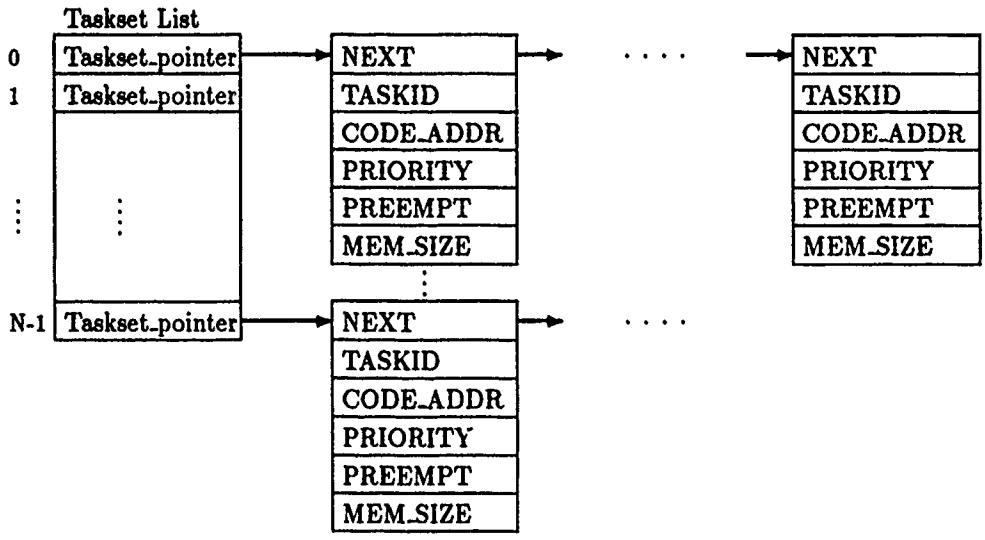


Figure 6.3. Taskset List Implementation

Taskset selection is implemented as follows. The application designer builds a taskset list for each specific application. The taskset list is an array of linked lists, with an aray element for each processor in the system. The linked lists connect *task headers*, data structures defining the initial configuration information for a single task. Each application task has a unique header; each header is in one of the linked lists. The taskset list implementation is shown in Figure 6.3. The SELECT LOCAL TASKSET routine selects the appropriate taskset for a processor, given the processor's iPSC/2 node ID. The taskset is then loaded as described in Section 6.3.1.2.3.

While this approach and the ARTMOS tasking model do not explicitly support dynamic task allocation, they could be easily expanded to provide support for reconfiguration of tasks in case of processor failure or to switch between application modes. The application designer can provide multiple taskset lists for different modes or number of processors. For example, a 20 task application could be broken into 5 parts in one taskset list and 4 parts in another, allowing a processor to select a taskset based on its node ID and how many nodes are active.

6.3.1.2 Task Management Many of the independent processes identified in Section 5.3.4 deal with the scheduling and manipulation of application tasks. Other task manipulation operations are performed by the operating system during initialization. All together, five operations are defined as acting on task objects:

1. create
2. Delay
3. Delay Until
4. Schedule
5. Terminate

Of the five, *create* is an internal ARTMOS operation, while the other four were defined as independent processes in Section 5.3.4. The last three are also system calls that may be invoked by the tasks themselves. This section describes the nature of the tasks, how they are managed by the operating system, and the task operations.

6.3.1.2.1 Task Control Blocks Section 4.3.4 introduced the notion that the operating system does not physically manipulate task objects. The *task* is a *logical* entity, represented in the operating system by a physical data structure called a Task Control Block (TCB), which stores all task state information when the task is not executing. Changes in the task state are made to the TCB object; manipulation of tasks involves manipulation of their associated TCBs.

The TCB object is extremely dependent upon the specific hardware because it records the complete state of the hardware when the task is not executing. With this hardware dependence, the TCB object will most likely change in structure between implementations and target systems. Also, as the OS functionality grows or the application changes, the TCB may need to adapt. It is therefore desireable to keep the details of the TCB structure hidden from the rest of the system.

By abstracting the TCB object and providing primitives to perform all necessary operations on the TCB, specific details of the TCB implementation can be hidden from the rest of the operating system. In this manner, changes to the TCB structures only

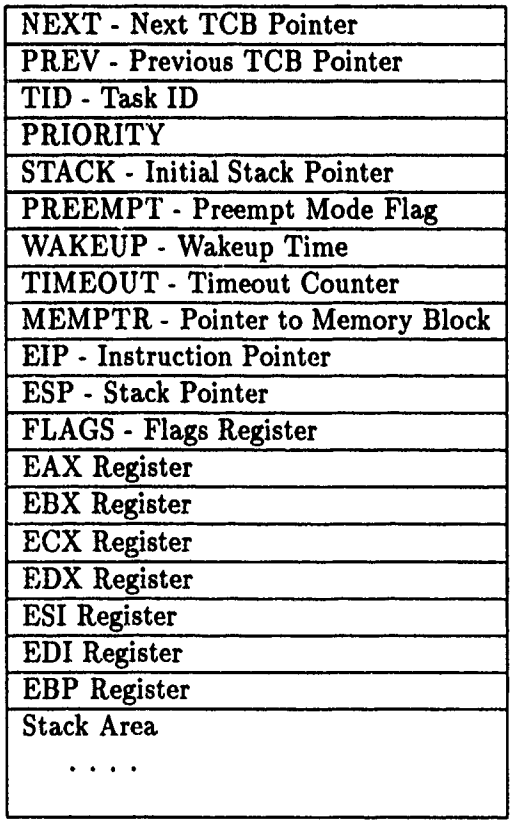


Figure 6.4. Task Control Block Definition

require modifications to the operating system primitives that manipulate the TCB. The *task* manipulation operations, listed above, do not need to be changed since they only affect the TCB through the TCB primitives. These primitives should provide the capability to read or set any of the fields in the TCB.

Based on the structure of the TCB shown in Figure 6.4, TCB operations needed by the ARTMOS include:

1. SAVE TASK CONTEXT
2. RESTORE TASK CONTEXT
3. SET INITIAL TASK CONTEXT
4. SET WAKEUP TIME
5. READ WAKEUP TIME
6. SET TIMEOUT

7. READ TIMEOUT
8. SET PREEMPT MODE (CHANGE PREEMPT MODE)
9. READ PREEMPT MODE
10. SET PRIORITY (CHANGE PRIORITY)
11. READ PRIORITY (GET PRIORITY)
12. SET TASKID
13. SET MEMORY POINTER

Operations followed by a second name in parentheses serve as both TCB primitives used by the ARTMOS and system calls for the application. The system call name is in parentheses. The system call handlers simply call the appropriate TCB manipulation routine.

The above operations are self explanatory and are not decomposed any further. The SET INITIAL TASK CONTEXT and SAVE/RESTORE TASK CONTEXT operations by their very nature are extremely hardware dependent and will be different for each target processor. For example, this design is based on the Intel 80386 CPU used in the iPSC/2, so the register fields within the TCB match those of the 80386. If a different processor were used, the TCB would reflect the registers of that processor. The other routines should not change much between implementations since they are based on the defined operating system data structure rather than a hardware device.

The TCB primitives listed above operate on an individual TCB. They do not affect the movement and management of TCBs or their states. These capabilities are discussed next.

6.3.1.2.2 Task Management Lists As application tasks move through different phases of their operation, they pass through different logical *states*, as shown in Figure 6.5. The transition of logical tasks between logical states is an abstraction, however; it is the physical TCB data structures that change state. Some mechanism is needed to manage the transition of TCBs between states and to represent which state a TCB (and its associated task) is in. This could be done with a status flag in the TCB or by maintaining lists with the TCBs for all tasks in a given state. The latter is the approach taken in the ARTMOS design.

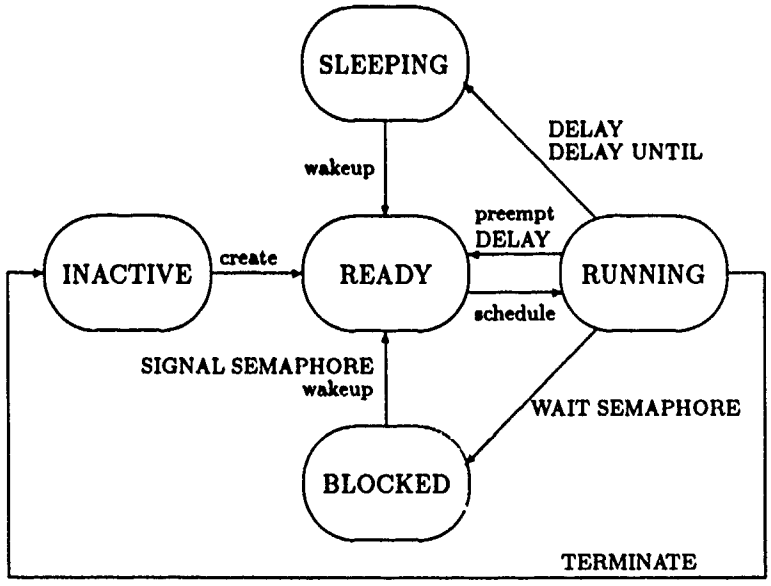


Figure 6.5. Task States

The SADT diagrams in the requirements specification document (located in Appendix A) identified several "TCB management lists" to "hold" the tasks when they were in the different states. The lists are the ready task list, the sleeping task list, and the blocked task (semaphore) lists. The nature of the lists was not defined, for at that stage, the manner in which the tasks were managed was not important. These "lists" were essentially "generic data structures," with no implied structure or organization to the lists or the TCB elements of the lists. The previous section defined the structure of the TCB object. The TCB management lists are now examined in greater detail.

To manage groups of TCB items, a simple linked list or queue structure can be used. This facilitates addition and removal of items according to various queue-ordering criteria. Figure 6.6 depicts a sample TCB queue. To implement these structures, either dynamic memory management is needed to provide new TCBs for the queues, or a static number must be defined in advance. For speed and simplicity, the latter approach is selected. Some number of TCBs are statically defined at initialization and stored in a "free TCB queue," from which TCBs can be obtained as needed.

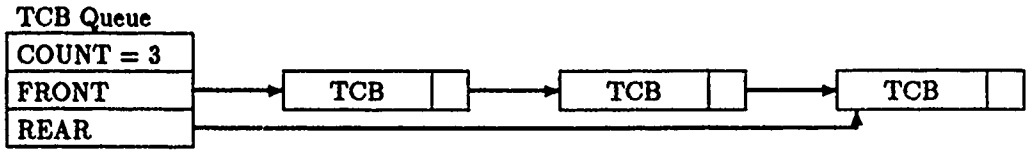


Figure 6.6. Task Control Block Queue

Since the TCBs in the free TCB queue contain no useful information, they require no particular ordering. This being the case, the queue should be ordered in such a way as to minimize the time to add and remove TCBs. A simple FIFO queue allows items to be added at the rear of the queue and removed from the front. TCBs are added via the PUT FREE TCB QUEUE operation and removed with GET FREE TCB QUEUE.

The ready queue poses some complicated questions, because its implementation is intricately tied to the scheduling algorithm selected. If the ready queue is FIFO, for example, a priority-based scheduler would be forced to search the entire list of ready tasks to find the one with the highest priority — clearly not an efficient approach. The ready queue implementation is therefore discussed in the section on task scheduling below. Regardless of the selected implementation, however, operations on the ready queue can be defined which make the implementation details transparent to the rest of the system. Changes to the queue algorithm requires changes only to the queue routines. The ready queue operations are: PUT READY QUEUE to add to the queue, GET READY QUEUE to remove from the queue, IS READY QUEUE EMPTY to check the queue status, and POINT FRONT READY QUEUE to obtain a pointer to the first task in the queue.

The sleep queue holds tasks which are delaying until a specific processor time. Since the tasks with the earliest wakeup times are removed from the queue first, the sleep queue should be ordered by the task wakeup times. The task due to wake up first is always at the front of the queue; the task with the most distant wakeup time is at the rear. New tasks are inserted at the appropriate place in the queue, based on their wakeup time. Tasks with the same wakeup time are ordered FIFO, since they are all removed when that time arrives.

The needed queue operations match those for the ready queue: PUT SLEEP QUEUE, GET READY QUEUE, IS SLEEP QUEUE EMPTY, and POINT FRONT SLEEP QUEUE.

The organization of the semaphore queues is described in detail below. Since a number of different schemes could be used to order the semaphore queues, and since the queue manipulation routines need to accept a pointer to the target semaphore queue as a parameter, specific operations are provided for the semaphore queues. As with the ready and sleep queues, operations are provided to add TCBs to the queues (PUT SEMAPHORE QUEUE), remove TCBs (GET SEMAPHORE QUEUE), determine if the queue is empty (IS SEMAPHORE QUEUE EMPTY), and return a pointer to the front of the queue (POINT FRONT SEMAPHORE QUEUE).

One final point can be made about the ARTMOS queue operations. The implementation of many of the queues, such as the ready and sleep queues, has been abstracted from the rest of the operating system by providing specific queue operations for those queues. An example is GET READY QUEUE. At a lower level, these capabilities could be provided by a small set of generic queue routines, such as GET FIFO QUEUE or PUT PRIORITY QUEUE. The "higher" level GET READY QUEUE or PUT SEMAPHORE QUEUE could simply call these underlying generic routines — making the code more efficient.

6.3.1.2.3 Task Operations The above discussion took a "bottom up" approach by examining the low-level data structures and operations needed to represent and manipulate TCBs. The high level *task* manipulation operations are built upon this foundation. These operations, introduced above, are: create, delay, delay until, schedule, and terminate.

Creating a Process During slave processor intialization, described in Section 6.3.1.1, the operating system on each processor selects a set of tasks to execute on that processor. These tasks are initially in a dormant or *inactive* state; the operating system "creates" the tasks by building a TCB for each task and loading them into the ready queue. This occurs in the LOAD TASKSET routine.

The creation process consists of obtaining an unused TCB and initializing it as appropriate for the task being created, as shown in Figure 6.7. Available TCBs are obtained

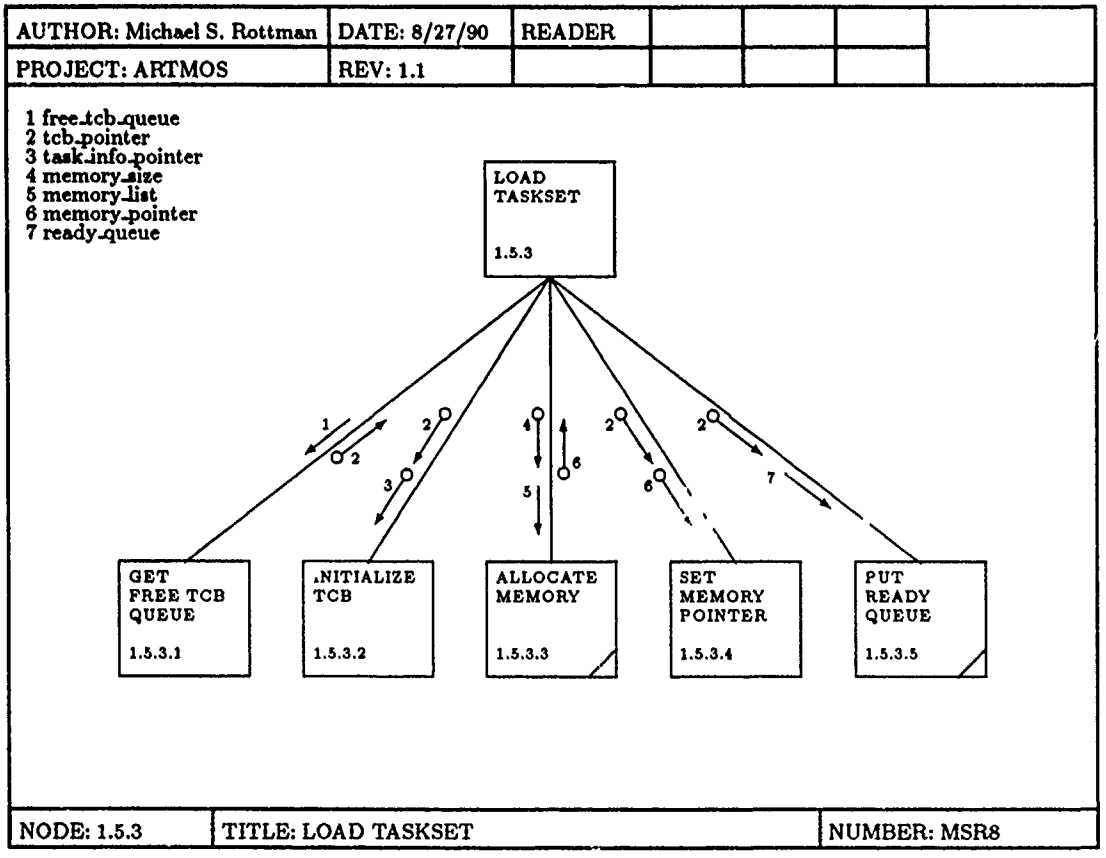


Figure 6.7. LOAD TASKSET Routine

from the free tcb queue u. g the GET FREE TCB QUEUE primitive. Since the available TCBs are statically declared during initialization, the application must not assign more tasks to a processor than the maximum number of TCBs. Correct application operation cannot be guaranteed if more tasks are assigned to a processor than the maximum number permitted.

The unused TCB is then initialized for the specific task being created. Some of the TCB elements are initialized identically for all tasks, such as the context of the processor and the timeout and wakeup fields being cleared. Other fields must be initialized differently for each task. To provide this task-dependent information, each task has an associated *task header* which describes the initial configuration of that task. The task header consists of fields defining the task's identification number, starting address of its code, initial priority, initial preempt mode, and memory size needs. This information, except for the memory

```
DELAY TASK (duration);
    MASK INTERRUPTS;
    IF (duration > 0) THEN
        wakeup time = time + duration;
        SET WAKEUP TIME (running task, wakeup time);
        SAVE TASK CONTEXT;
        PUT SLEEP QUEUE (running task);
    ELSE
        SAVE TASK CONTEXT
        PUT READY QUEUE (running task);
    ENDIF;
    SCHEDULE TASK;
END;
```

Figure 6.8. DELAY Algorithm

size, is loaded into the appropriate fields in the TCB. The memory size is passed to the memory allocation routine to obtain a block of memory for the task. A pointer to the memory block is placed into the TCB. The task is then loaded into the ready queue.

Delaying a Task The ARTMOS provides two operations which allow the design and implementation of periodic tasks: DELAY TASK and DELAY TASK UNTIL. These functions allow a task to "go to sleep" for an interval of time or until a specific time, respectively. The operations are similar, but have several important differences.

If a task calls DELAY TASK with a duration greater than zero, the timeout field in the task's TCB is set to the current time plus the duration value, its context is saved, and it is added to the sleep queue. A duration of zero (or less), however, is a special case used when the task wishes to release the processor but not go to sleep. If the task is waiting for some event, for example, it can allow other tasks of equal priority to use the processor while it is waiting. For this case, the calling task's context is saved and the task is put back in the ready queue. Figure 6.8 lists the algorithm for this operation.

The DELAY TASK UNTIL operation is passed an absolute time at which the calling task is to be awakened. A wakeup time less than or equal to the current time is considered

to be an error, since it indicates that the timing of the calling task is incorrect. In this case, an error flag is returned to the task and the delay operation is not performed. If the wakeup time is greater than the current time, the wakeup time field is set in the task's TCB, its context is saved, and the task is added to the sleep queue.

The saving of a task's context is achieved via the SAVE TASK CONTEXT operation. This routine places the values of the processor hardware registers into the appropriate components of the TCB (Figure 6.4). The registers reflect the exact state of the processor at the time the task is switched out, and consist of the instruction pointer, stack pointer, flags register, and the general purpose registers.

Both the DELAY TASK and DELAY TASK UNTIL operations invoke the ARTMOS scheduler after the calling task has been put in the sleep queue. The scheduler is described next.

Scheduling a Task The determination of which ready task to execute at any given time is called *scheduling* (6:109), (56:113). There are many different scheduling policies that can be used, including: first come first serve, shortest job first, priority, round robin, shortest remaining time (6:109),(56:113), and multi-level (57). In many cases, new tasks may cause the running task to be preempted, based on some criteria (such as priority, execution time, and so on).

Most of the scheduling algorithms described above derive from time sharing systems, which tend to schedule tasks and allocate resources by criteria such as fairness, throughput, and response time rather than timeliness. This is not appropriate for real time systems (75:17–18), (52:13–17). Scheduling theory must focus on meeting the specific timing requirements of real-time systems so that the timing behavior of the system is predictable and maintainable. One such scheduling scheme, rate-monotonic scheduling (RMS) (68), is a technique for assigning priorities to real-time tasks in decreasing order of task execution period. The most frequently executed task gets the highest priority, and the least frequently executed task gets the lowest priority. This approach provides a means of proving that the design meets hard deadlines for critical tasks and for predicting the behavior of the system overload conditions.

The use of simple static priorities for task scheduling is questioned by Stankovic (74:4) and Northcutt (52:13). They note that real-time tasks are characterized by two unrelated criteria: the *urgency* of the task and the *importance* of the task relative to other tasks. When priority driven scheduiing is used, urgency and importance must be mapped into one factor — priority. This approach requires many iterations to identify the proper priorities and denies the opportunity to adapt to run-time changes. Further research is required in this area.

Which scheduling algorithm is used for the ARTMOS design? To keep the ARTMOS design as simple as possible while still meeting the initial requirements, a priority-based scheduler is used. That is, the executing task is always the ready task with the highest priority. Because many of the operations performed by the operating system during interrupt servicing or in response to system calls can impact the schedule, the selected algorithm is preemptive. If while a task is executing a higher priority task becomes ready, the running task is *preempted:* its context is saved and the task is put back in the ready queue. The higher priority task is then given the processor.

It should be noted that the intent of this investigation is not to determine the "best" or most appropriate scheduling algorithm for given real-time or parallel systems. Theoretically, any scheduling algorithm capable of meeting the real-time constraints of the applications could be used in the ARTMOS. Indeed, the ARTMOS design applies the software engineering principles of abstraction and information hiding to insulate the scheduling algorithm from the rest of the operating system. Moving to a different scheduler requires different scheduling routines, but no other changes.

The efficiency of the scheduling algorithm can be significantly impacted by the ready queue implementation. For ARTMOS, with a priority-driven, preemptive scheduler, the ready queue should be ordered by priority: the highest priority ready task is always at the front. To schedule a task, the scheduler simply removes the front task from the ready queue. No time is wasted searching the queue for the highest priority task. If many tasks are waiting, time may be wasted searching for the appropriate position to insert a new task. One solution is to implement the ready queue as a set of queues, one for each priority or group of priorities. This is done in the UNIX and Mach operating systems (7). For

simplicity, however, the ARTMOS uses a single, priority-ordered ready queue. The actual queue organization would matter only to the ready queue handler routines GET READY QUEUE, PUT READY QUEUE, IS READY QUEUE EMPTY, and POINT FRONT READY QUEUE. These routines are the means by which the higher level scheduling operations manipulate the ready queue.

Task scheduling is performed by two independent processes identified in Section 5.3.4, CHECK SCHEDULE and SCHEDULE TASK. To understand the function and interaction of these operations, the manner in which they may be invoked must be considered. Some operations, such as the TERMINATE system call, result in the running task being switched out, or in this case, removed from the system entirely. A new task must be selected from those waiting in the ready queue. In other cases, an operation alters the state of the running task or adds new tasks to the ready queue. The CHANGE PRIORITY system call, for example, modifies the priority of the calling task. The clock interrupt increments the local clock, then wakes any tasks delaying until the new time. In both cases, the running task may not have the highest priority any longer and so the ready queue must be checked to determine whether a higher priority task is ready.

SCHEDULE TASK, as the name implies, selects the highest priority ready task and allocates the processor to that task. This is accomplished by calling GET READY QUEUE to pick the first task from the ready queue, then restoring the task's hardware state via the RESTORE TASK CONTEXT routine. RESTORE TASK CONTEXT removes the contents of the TCB register components and places them in the actual processor registers so that the task can resume executing as if never interrupted. A call to SCHEDULE TASK never returns, since control is eventually passed to the highest priority task.

CHECK SCHEDULE is invoked when the ready queue has been altered to determine whether or not the running task should be preempted. This decision is made by comparing the priority of the front task in the ready queue to that of the running task. If the running task has the higher priority, control is passed back to that task. When the running task must be preempted, its context is saved with the SAVE TASK CONTEXT routine and it is put into the ready queue with PUT READY QUEUE. SCHEDULE TASK is then invoked to start the new highest priority task.

Terminating a Process ARTMOS provides a system call which allows application tasks to inform the operating system when they have completed their purpose. Many tasks are periodic in nature and thus never finish, executing continuously as an endless loop via the DELAY or DELAY UNTIL operations. Other tasks may be needed only once for special conditions, or may loop only for a period of time or in a certain mode. At some point, these tasks finish. Rather then remain active for no reason and using processor resources, the task issues a TERMINATE system call. When invoked by this system call, the TERMINATE TASK process deallocates the memory assigned to the calling task, then returns the TCB to the free tcb queue for reutilization.

6.3.1.3 Communications This section describes the design of the ARTMOS intertask communications facilities, specified in Section 4.3.6. Topics discussed include a review of the communications requirements, the mapping of these requirements to the iPSC/2 hardware architecture, the data structures used, and the design of the communications routines themselves. Consideration is also given to the manner in which tasks block when the communications protocol must wait for some event, such as a message to be produced. This question was introduced in Section 4.3.6, but could not be adequately answered until the design stage.

6.3.1.3.1 Requirements The ARTMOS communications model is based on that of the AMCAD RTMOS to comply with the objectives of this thesis investigation. As described in Section 1.1.3, AMCAD uses a shared memory model, with a unique mailbox structure defined for each global data variable. Application tasks communicate with one another through these global variables. The explicit requirements for the ARTMOS communications model are summarized below:

1. Tasks communicate through global variables.
2. Each global variable has exactly one producer and one consumer task.
3. Access to the global variables is controlled through producer and consumer flags associated with each variable to protect the integrity of data exchanges.
4. Use of the producer/consumer algorithm is transparent to the calling tasks.

5. Global variables are not implemented as queues, in which new data is added to the queue for the consumer to eventually read. Rather, producing a global variable overwrites the previous data in the variable. Consumer tasks always receive the most recent data produced to the variable.
6. An asynchronous SEND MESSAGE primitive produces a data variable immediately regardless of whether the previous data has been consumed.
7. A synchronous SEND MESSAGE AND WAIT primitive waits for the previous data to be consumed before producing new data.
8. A RECEIVE MESSAGE primitive consumes a global variable. If the variable has not been produced since the last consumption, RECEIVE MESSAGE waits until data is produced.
9. The wait times in SEND MESSAGE AND WAIT and RECEIVE MESSAGE are limited to a period of time specified by the calling task.

6.3.1.3.2 Implementation of Shared Memory This section considers the implementation of the specified communication model on the target machine, the Intel iPSC/2. Several characteristics of the iPSC/2 architecture impact the design of communications algorithms. The iPSC/2 is a hypercube architecture, composed of $N = 2^n$ processors interconnected as an n-dimensional binary cube (18:119). Each processor is directly linked to its n nearest neighbor processors along the vertices of the cube. The iPSC/2 uses Direct-Connect routing modules (DCM) on each node to implement a circuit switched network and interface the nodes to the network (18:441). The DCM establishes a circuit or path between nodes, In addition, only the source and destination nodes are involved; intermediate nodes along the established path are not interrupted. Because of the direct connection of source and destination nodes, broadcasting requires n steps for an n-dimensional cube. The sending node sends the data to a neighbor, they both send to another neighbor, these four send to their neighbors, and so on (37:4.11). Half the nodes participate in sending the message. The challenge for the ARTMOS is to implement a shared memory communications model on this message passing architecture. Three approaches are possible.

One possible approach is to designate one node as the shared memory. That node is responsible for maintaining the shared memory variables and manipulating the producer and consumer flags. Producer tasks send a data message to the shared memory node, which updates the global variable and the producer flag. Tasks needing data request a specific

global variable from the shared memory node. The shared memory node responds by sending the global data to the requesting node, if the variable had been produced since last consumed. This approach requires much less memory on the nodes, since shared memory is not replicated. The iPSC/2 direct-connect hardware makes communications time between the shared memory node and the other nodes consistent, regardless of which nodes are communicating. The disadvantages far outweigh the advantages, however. Depending on the number of nodes in the system, using one node as shared memory could cause a bottleneck. Further, additional software would be required on the shared memory node to implement a protocol for interacting with the other nodes. Finally, since the other nodes would not have local access to the producer/consumer flags, timeouts for the receive and synchronous send operations would have to be performed at the shared memory node. The receiving node would provide a timeout value with the request, then block until some response is received from the shared memory node.

A second method is to physically replicate the shared memory on each node (43, 85). The global variables are declared on every node. A task producing data broadcasts it to the "shared memory" on the other nodes. Upon receiving the broadcast, nodes update their local shared memory and set the producer flag. Tasks consume data from their node's copy of the shared memory, then broadcast the updated consumer flag. This approach is simple and avoids the shared memory bottleneck. The producer/consumer algorithm is faster and easier to implement, since each node has a local copy of the flags to manipulate, and communications is independent of node-task assignment. Though producing data involves formatting and sending a message, all data consumption is local, greatly reducing the receive time, the most time critical. This approach also provides the most flexibility for reconfiguration of the software task load. The most critical disadvantage of physically replicating the shared memory is the need to broadcast all data. All nodes are interrupted to receive, process, and possibly send each message, whether or not a task on that node needs the data. Broadcasting thus ties up the communications network needlessly while causing variable length delays between when different nodes receive the data. A further asymmetry is added because the originating node does not receive the message (37:4.11); it must copy the data into its local variable and set the flags.

The third approach is similar to the second, but without the disadvantages of broadcast communications. All global variables are declared on each node, as before, but data messages are only routed to the nodes with the tasks that consume them rather than broadcasting them to all nodes. A task producing a global variable performs a SEND MESSAGE operation, which updates that variables producer/consumer flags in the local copy of the variable and sends the data message directly to the node with the consumer of the variable. When the message is received at the destination node, that node's global variable and flags are updated with the new data. On any given node, only the variables needed by the tasks on that node are used. This approach combines the advantages of the previous two approaches. The producer/consumer flags are locally accessible to the tasks that need them, data is locally available to the consumer task, and sending can be optimized when the consumer and producer tasks are assigned to the same node. Though this approach uses point-to-point message passing to approximate shared memory, the application views communications as if a physical shared memory were present. The disadvantages are that the global memory is used inefficiently and that communications is no longer independent of the task assignment. Each node must have some record of where the tasks are assigned in order to route data to the appropriate nodes.

Approach three is used for the ARTMOS communication design. It makes efficient use of the iPSC/2 communications architecture and provides a good approximation of shared memory. The producer/consumer algorithm can be implemented in a straightforward manner since each node has local access to the needed flags. Further, RECEIVE MESSAGE becomes a *local* operation, thereby minimizing the time to provide the application task with the data it needs since off-node communication is needed. Though global memory is used inefficiently with this approach, two factors must be considered. First, the global variables need the same physical address on each node so the message received at a node can be routed to the correct variable. Second, this approach provides the flexibility to enable task reconfiguration since the data structures won't have to be moved or redeclared. The other disadvantage to this approach is that nodes need to know where tasks are assigned in order to route messages to the correct nodes. While this is true, the iPSC/2 DCM hardware provides near-uniform transmission time whether the message is sent be-

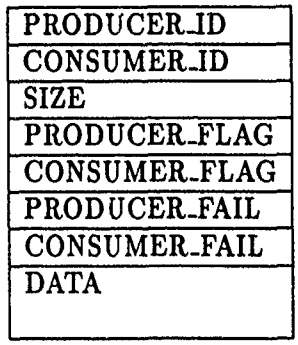


Figure 6.9. Global Variable Structure

tween nearest neighbors or the most distant nodes (18:441). Computational efficiency is essentially independent of the task-to-node assignment.

6.3.1.3.3 Producer/Consumer Algorithm The global data variables introduced above serve as "mailboxes" through which producer and consumer tasks communicate. A unique mailbox variable is defined for each instance of intertask communications; that is, each data item passed between tasks has a unique global variable through which the data is passed. When tasks produce data needed by another task, the data is sent via operating system primitives (SEND MESSAGE and SEND MESSAGE AND WAIT) to the data item's global variable. The task needing this data item uses another operating system primitive (RECEIVE MESSAGE) to consume the data in the global variable.

The structure of this variable, with which the producer/consumer algorithm is implemented, is shown in Figure 6.9. The PRODUCER ID and the CONSUMER ID fields are the ID numbers of the tasks authorized to produce and consume the variable, respectively. These prevent a variable from being corrupted by an unauthorized task. SIZE is the number of bytes of data in the global variable. The next four fields, PRODUCER FLAG, CONSUMER FLAG, PRODUCER FAIL, and CONSUMER FAIL, are used by the producer/consumer algorithm to record the number of successful and failed productions and consumptions. The data item is stored in the DATA area of the global variable.

The producer/consumer algorithm is the heart of the ARTMOS intertask communications facility, as can be seen from the requirements summarized in Section 6.3.1.3.1.

The algorithm uses the producer and consumer flags in a global variable to protect access to that variable and to indicate the status of the data in the variable. All decisions about what to do when producing new data and consuming data are based on this algorithm and these flags.

The PRODUCER FLAG and CONSUMER FLAG are essentially counting flags which indicate the number of successful productions and consumptions. When the PRODUCER FLAG is one greater than the CONSUMER FLAG, the variable has been successfully produced; if the flags are equal, the variable has been successfully consumed. A task attempting to consume a variable that has not yet been produced waits for a specified number of clock ticks for the PRODUCER FLAG to be set (incremented greater than the CONSUMER FLAG). If the task times out before the data is produced, the consume is considered a failure: the task is given the old data and the CONSUMER FAIL flag is incremented. A similar situation arises when a task produces a variable whose data has not been consumed. In this case, the old data is overwritten and the PRODUCER FAIL flag is incremented. The PRODUCER FLAG and CONSUMER FLAG are not incremented on failed productions or consumptions.

As stated in Section 6.3.1.3.1, the producer/consumer algorithm is embedded in the operating system communications primitives and is transparent to the application tasks. The ARTMOS communications primitives and support routines are examined next.

6.3.1.3.4 Data Structures As discussed in Section 6.3.1.3.2, the shared memory communications model as implemented on the iPSC/2 requires that data be produced globally and consumed locally. That is, when tasks produce data items, the items are transmitted to the processor node where the item's consumer task is assigned; data is consumed from the local copy of the global variable. A RECEIVE operation still performs a global send, however, to update the CONSUMER FLAG on the sending node's copy of the variable.

Because of this implementation, only two operations actually produce data messages to transmit between nodes: the SEND MESSAGE and SEND MESSAGE AND WAIT operating system primitives. The operations accept data item from tasks, building them

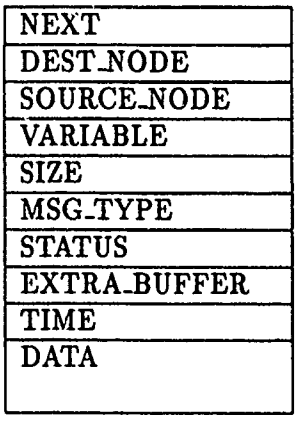


Figure 6.10. Datagram Structure

into data messages for transmission. The SEND CONSUMED FLAG routine used by the RECEIVE MESSAGE primitive is a special case of the others, in that it generates messages contains only status flags, no data. These messages between nodes are represented by *datagrams*.

The datagram data structure (Figure 6.10) contains all information the operating system needs to build and process data messages. The NEXT field, not sent with the datagram, can be used to construct a linked list of datagram objects. DEST NODE and SOURCE NODE record the destination and source nodes of the datagram, respectively, and are used to route the message over the iPSC/2 communications network. VARIABLE is a pointer to the global memory variable to which the data is written. The SIZE field is the number of data bytes being sent, and can be zero if the message is used to signal some event or to send the consumed flag. The MSG TYPE is either *message* or *response*. Response datagrams are generated by SEND CONSUMED FLAG to update the global variable on the source node. The STATUS field provides the RECEIVE INTERRUPT HANDLER with information concerning the success or failure of the send or receive. For example, if data is produced successfully, the STATUS field is PRODUCED, to signal the interrupt handler to increment the PRODUCER FLAG in the global variable on the destination node. EXTRA BUFFER can point to an additional data buffer, if needed, to supplement the datagram DATA area. TIME is the clock time on the source node when the message is sent. The data item is stored in the DATA field.

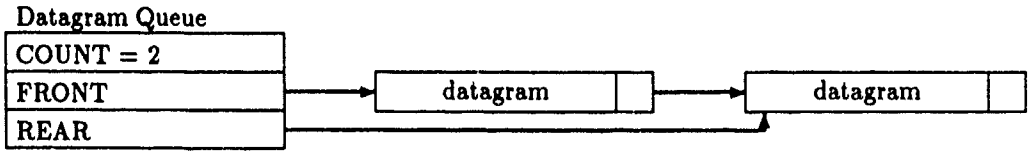


Figure 6.11. Sample Datagram Queue

Three data structures are used to manage datagrams for internode communications: the free receive queue, free send queue, and send queue. Figure 6.11 depicts a sample datagram queue. The free receive and free send queues hold free datagrams that are used to receive and send datagrams, respectively. These queues are setup during initialization of the local processor data structures as linked lists of available datagrams with fixed size data buffers. The receive datagrams are used by the receiver hardware to store incoming messages. The free receive queue therefore has a small number of datagrams with buffers the size of the largest possible datagram since the node must have a large enough buffer ready to load the incoming message.

The free send datagrams are used to build messages to transmit to other nodes, and so the free send queue must contain a large number of datagrams to make sure a task doesn't have to wait for a datagram to be freed up. The fixed size data buffers in send datagrams are small for efficient memory use. If a task wishes to send a message larger than the default send buffer, the memory allocation routine is invoked to acquire a supplemental buffer. A pointer to this buffer is placed in the EXTRA BUFFER field of the datagram. The exact default size of the buffers should be determined by the average message size of the application, to make the send primitives as fast as possible and avoid the need to request an additional buffer.

The send queue is a data structure which serves as a storage area for datagrams that cannot be sent yet because the transmitter hardware is busy. This structure is organized as a FIFO queue. The PUT DATAGRAM QUEUE operation adds messages to the rear of the queue, and the GET DATAGRAM QUEUE operation removes messages from the

front. Datagrams are therefore transmitted in the order in which they arrive. If it is determined that messages should be transmitted according to the priority of the sending task, the two queue manipulation routines would be the only code that needed to be altered to accommodate the change. Note, however, that GET and PUT DATAGRAM QUEUE are also used to access the free send and receive queues, which are also FIFO.

6.3.1.3.5 Communications Protocol Application tasks have three operating system primitives (system calls) for intertask communications, as specified in Section 6.3.1.3.1: SEND MESSAGE, SEND MESSAGE AND WAIT, and RECEIVE MESSAGE. This section describes the operation of these routines, plus several needed interrupt handlers and support routines.

The SEND MESSAGE and SEND MESSAGE AND WAIT system calls allow tasks to produce data items to their associated global variables. Both system calls accept a data item, the size of the data, and a pointer to a destination global variable, from which they build a datagram to transmit the item to the destination node. The difference between the two calls is that SEND MESSAGE overwrites the old data in the global variable, regardless of whether that data has been consumed; SEND MESSAGE AND WAIT waits a specified number of clock ticks for the old data to be consumed before overwriting the old data. As ould be expected, the SEND MESSAGE AND WAIT algorithm is identical to that of SEND MESSAGE except for an additional loop for the timeout. This discussion focuses on the SEND MESSAGE AND WAIT algorithm, shown in Figure 6.12, since it is a superset of the SEND MESSAGE algorithm.

The send algorithms first check whether the task attempting to produce the global variable is authorized to do so. Each variable has fields identified the tasks authorized to produce and consume that variable. If the ID of the producer task does not match the variable's PRODUCER ID field, control is returned to the task with an "invalid sender" status flag. For valid producer tasks, SEND MESSAGE AND WAIT enters a loop waiting for either the previous data to be consumed or the timeout to expire. The algorithm releases the processor after each unsuccessful time through the polling loop by issuing a DELAY TASK system call with a delay duration of zero. This puts the sending task

```
SEND MESSAGE AND WAIT (local var ptr, global var ptr, size, timeout, status)
   IF (running task.TASKID ≠ global var ptr.PRODUCER ID) THEN
      status = invalid sender;
   ELSE
      status = data unconsumed;
      WHILE (status = data unconsumed) DO
         IF (global var ptr.PRODUCER FLAG = global var ptr.CONSUMER FLAG) THEN
            status = OK;
            increment global var ptr.PRODUCER FLAG;
         ELSE IF (timeout > time) THEN
            DELAY TASK (zero);
         ELSE
            status = previous data overwritten;
            increment global var ptr.PRODUCER FAIL;
         ENDIF;
      ENDDO;
      MASK INTERRUPTS;
      GET NODE ID (global var ptr.CONSUMER ID, consumer node);
      IF (consumer node = this node) THEN
         COPY BUFFER (local var ptr, global var ptr.DATA, size);
      ELSE
         BUILD DATAGRAM (local var ptr, global var ptr, size, consumer node, data,
         datagram ptr);
         IF (status = OK) THEN
            datagram ptr.STATUS = PRODUCE;
         ELSE
            datagram ptr.STATUS = OVERWRITE;
         ENDIF;
         IF (send datagram = NULL) THEN
            send datagram = datagram ptr;
            SEND HANDLER (send datagram);
         ELSE
            PUT DATAGRAM QUEUE (send queue, datagram ptr);
         ENDIF;
      ENDIF;
      UNMASK INTERRUPTS;
END;
```

Figure 6.12. Send Message and Wait System Call

back in the ready queue, allowing other ready tasks of equal priority a chance to execute while the sending task is waiting. The local copy of the variable's PRODUCER FLAG or PRODUCER FAIL field is updated in the loop when the associated event occurs, dropping the algorithm out of the polling loop.

The next step in the send algorithms, identified in Section 6.3.1.3.2, is to determine the processor node on which the global variable's consumer task is running. This could be implemented many different ways, such as scanning the taskset assignment lists until the indicated task is found, or by simple table lookup. The particular method used is not important, though it should be fast and predictable to avoid messing up the timing of the send algorithm. To isolate the send algorithms from the selected method, a routine is called (GET NODE ID) with the consumer task ID which returns the task's processor node ID. If the consuming node is the producing node, the data item is copied directly into the local copy of the global variable.

When the consumer node is *not* the producing node, a datagram must be "built" to send the data item between the two nodes. This process is performed by a routine called BUILD DATAGRAM (Figure 6.13). BUILD DATAGRAM gets a datagram from

```
BUILD DATAGRAM (local var ptr, global var ptr, size, node, type, datagram ptr);
    GET DATAGRAM QUEUE (free send queue, datagram ptr);
    IF (size > send buffer size) THEN
        ALLOCATE MEMORY (size – send buffer size, memory pointer);
        datagram ptr.EXTRA BUFFER = memory pointer;
        COPY SEND BUFFER (local var ptr, datagram ptr, size);
    ELSE IF (size > zero) THEN
        COPY BUFFER (local var ptr, datagram ptr.DATA, size);
    ENDIF;
    datagram ptr.DEST NODE = node;
    datagram ptr.SOURCE NODE = this node;
    datagram ptr.VARIABLE = global var ptr;
    datagram ptr.MSG TYPE = type;
    datagram ptr.SIZE = size;
    datagram ptr.TIME = time;
END;
```

Figure 6.13. Build Datagram Routine

the free send queue, then determines whether the default send buffer size is large enough to hold the data item being sent. If so, the data item is copied into the datagram data area; otherwise, additional memory is allocated to serve as an extra buffer. The datagram is loaded with as much of the data item as it can hold, with the remainder copied into the extra buffer. BUILD DATAGRAM then sets up the remaining datagram fields: the source and destination node fields, the pointer to the global variable, the message type, and the size of the data item. Lastly, the current time is loaded into the datagram TIME field.

Once the datagram is built, SEND MESSAGE AND WAIT sets the STATUS field to PRODUCE or OVERWRITE, based on the results of the earlier polling loop. This field is used on the receiving node to set that node's copy of the variable flags the same as they are on the sending node. Finally, the algorithm attempts to transmit the datagram. If the transmitter is busy, the datagram is placed on the send queue. If the transmitter is not busy, the datagram is passed to the SEND HANDLER routine for transmission. Transmitter status is indicated by a variable, *send datagram*, which points to the datagram currently being sent. If send datagram is null, the transmitter is idle.

SEND HANDLER is provided to isolate the sending primitives from the specifics of the iPSC/2 transmitter hardware. Once SEND MESSAGE, SEND MESSAGE AND WAIT, or SEND CONSUMED FLAG obtained a free datagram and load the datagram with the message information, they call SEND HANDLER with a pointer to the datagram as a parameter. SEND HANDLER performs all the hardware-specific actions needed to actually transmit the datagram to the destination node. This includes computing the routing tags used to set up the datagram's path through the network, interacting with the DCM, and initiating the data transfer. Changes to the underlying communications hardware can thus be made without affecting the operating send primitives.

The iPSC/2 transmitter interrupts when it finishes transferring a datagram across the bus (37:4.10). The SEND INTERRUPT HANDLER, shown in Figure 6.14, first resets the transmitter hardware so it is ready for another interrupt. The datagram just sent is recycled by putting it back in the free send queue and the extra buffer memory (if used) is deallocated. If any datagrams are waiting in the send queue, the datagram at the front of the queue is removed and passed to the SEND HANDLER for transmission.

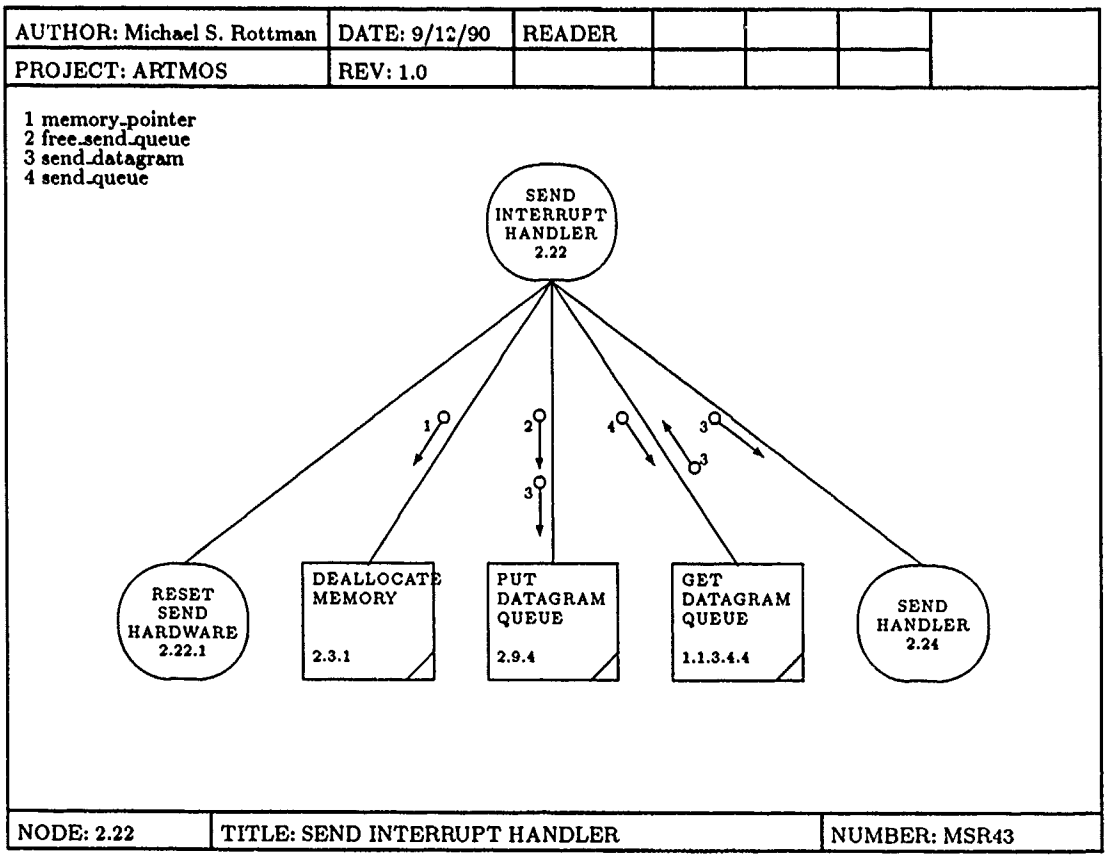


Figure 6.14. Send Interrupt Handler

ARTMOS always keeps the receiver setup to load a message into a fresh receive datagram. When a datagram arrives at the destination node and has been loaded into the datagram, the receiver interrupts and invokes the RECEIVE INTERRUPT HANDLER (Figure 6.15). This handler first acquires a new free receive datagram and resetting the receiver hardware so it can accept a new transmission. Once the hardware is reset, the handler inspects the STATUS field of the received datagram to determine what to do next. If the received message is a response, it is signalling that the indicated global variable has been consumed. Based on whether the STATUS field is set to CONSUME or PREVIOUS CONSUMED, the variable's CONSUMER FLAG or CONSUMER FAIL fields are updated, respectively. For data messages, the data is copied from the datagram to the data area of the global variable and the PRODUCER FLAG is incremented if STATUS equals PRODUCE. When STATUS is OVERWRITE, the PRODUCER FAIL

```
RECEIVE INTERRUPT HANDLER;
    datagram ptr = receive datagram;
    GET DATAGRAM QUEUE (free receive queue, receive datagram);
    RESET RECEIVE HARDWARE;
    global var ptr = datagram ptr.VARIABLE;
    IF (datagram ptr.STATUS = CONSUME) THEN
        increment global var ptr.CONSUMER FLAG;
    ELSE IF (datagram ptr.STATUS = PREVIOUS CONSUMED) THEN
        increment global var ptr.CONSUMER FAIL;
    ELSE IF (datagram ptr.STATUS = PRODUCE) THEN
        COPY BUFFER (datagram ptr.DATA, global var ptr.DATA, global var ptr.SIZE);
        increment global var ptr.PRODUCER FLAG;
    ELSE IF (datagram ptr.STATUS = OVERWRITE) THEN
        COPY BUFFER (datagram ptr.DATA, global var ptr.DATA, global var ptr.SIZE);
        increment global var ptr.PRODUCER FAIL;
    ENDIF;
    COMPUTE CLOCK ERROR (datagram ptr.TIME);
    PUT DATAGRAM QUEUE (free receive queue, datagram ptr);
END;
```

Figure 6.15. Receive Interrupt Handler

field is incremented. With this protocol, the flags on the source and destination nodes remain consistent. When the appropriate flags are set and data copied, the datagram is put back in the free receive queue.

The RECEIVE MESSAGE system call allows application tasks to consume a global variable. The receive algorithm, shown in Figure 6.16, begins by checking the variable's CONSUMER ID field to make sure the calling task is authorized to consume the data variable; if not, the request is rejected. For valid consumer tasks, the algorithm performs a polling loop very similar to that of the SEND MESSAGE AND WAIT operation: it loops until either the variable is produced or the timeout expires. After each unsuccessful iteration, the processor is released using DELAY TASK, as described above. Each of the loop-ending events call a routine called SEND CONSUMED FLAG, which builds and sends a datagram of response type and data size zero to the producer node. This response datagram ensures that the global variable's flags on the producer node are consist with those on the consumer node. Once the loop completes and the CONSUMED flag is transmitted,

```
RECEIVE MESSAGE (global var ptr, local var ptr, size, timeout, status)
    IF (running task.TASKID ≠ global var ptr.CONSUMER ID) THEN
        status = invalid receiver;
    ELSE
        status = data unconsumed;
        WHILE (status = data unconsumed) DO
            IF (global var ptr.PRODUCER FLAG > global var ptr.CONSUMER FLAG)
            THEN
                status = OK;
                increment global var ptr.CONSUMER FLAG;
                SEND CONSUMED FLAG (global var ptr, CONSUMED);
            ELSE IF (timeout > time) THEN
                DELAY TASK (zero);
            ELSE
                status = previous data consumed;
                increment global var ptr.CONSUMER FAIL;
                SEND CONSUMED FLAG (global var ptr, CONSUMED PREVIOUS);
            ENDIF;
        ENDDO;
        size = global var size;
        COPY BUFFER (global var ptr.DATA, local var ptr, size);
    ENDIF;
END;
```

Figure 6.16. Receive Message System Call

the global variable's data is copied to the specified local variable (regardless of whether the data is old or new).

A comment must be made about the portability of the ARTMOS design. To enhance portability to other parallel architectures and to insulate ARTMOS from changes to the communications section, the communications protocol is designed to hide implementation details from the rest of the ARTMOS. This principal is applied inside the communications routines as well. For example, the only means of communicating between tasks is via the three operating system primitives. Tasks wishing to send data to another task must use one of the SEND primitives. The SEND routines build datagrams for transmission, but must call another routine, SEND HANDLER, to perform the actual interaction with the communications hardware. Tasks are insulated from the communications protocols by the communications primitives, and the primitives themselves are protected from specifics of

the hardware by the SEND HANDLER. Likewise, the SEND and RE' .VE interrupt handlers are isolated from hardware details by the RESET SEND HARDWARE and the RESET RECEIVE HARDWARE subroutines.

6.3.1.4 Memory Management Section 4.3.5 specified the memory management functions needed by the ARTMOS. The operating system allocates memory to application tasks for private variables and workspace when the tasks are created. Memory is deallocated when the task terminates. The tasks themselves cannot request or return additional memory. A further requirement was identified during the detailed design of the communications software. Fixed size buffers are defined during initialization for sending and receiving internode messages. If an application task attempts to send a message larger than the default send buffer size, additional memory is allocated to serve as a secondary send buffer.

6.3.1.4.1 Requirements The underlying assumption is that each node has sufficient memory for its maximum workload. A system with hard real-time constraints cannot afford to have critical tasks blocked because memory (or some other) resource is not available. Many real-time operating systems return error codes indicating that the request failed, or make sure all resources are available before scheduling the task. Neither approach addresses the *real* problem: hard real-time constraints cannot be met if tasks are forced to wait for indefinite periods of time for resources that are not available. The solution is to require that sufficient memory be available. There is always "enough" memory for the application.

This is not to say, however, that a badly designed or inappropriate memory management algorithm might not place memory in such a state that, though enough memory is available, no one section of available memory is large enough for the request. The memory management functions must be designed such that such a condition does not occur.

An additional requirements impacting the selection of a memory management algorithm is that the memory provided a task must be contiguous. The presence of a memory management unit cannot be assumed in a real-time embedded system, so the overhead of

managing non-contiguous memory would be forced on the operating system and the application tasks. Likewise, it is assumed that once a memory block is allocated to a task, that memory block cannot be moved or reallocated until the task terminates and the memory is deallocated.

6.3.1.4.2 Approach The specification called for a fixed size memory management scheme. The rationale was that managing memory as fixed size blocks is more predictable and has lower operating system overhead than dynamically allocating the exact of memory needed at run-time. One problem with fixed size block allocation algorithms, however, is that it is possible for a memory request to be refused, not because there is insufficient memory, but because none of the available blocks are big enough. Even if two adjacent blocks are available, they cannot be combined into a larger block because of the fixed partition allocation and deallocation algorithms. Likewise, multiple blocks can not be allocated to the task unless they are contiguous. Even so, the application would need to be able to manage the use of several memory blocks rather than just one. Further, a fixed partition allocation scheme uses memory inefficiently for providing buffers for transmitting messages, where the size of the buffer needed is completely dependent upon the size of the application messages. A variable size block allocation approach can thus provide more efficient use of the available memory while minimizing fragmentation.

6.3.1.4.3 Algorithm The data structure used to represent the status of memory is an integral part of the algorithms for allocation and deallocation. To shield ARTMOS from the effects of changing data structure representations, any request for memory must eventually invoke *allocate* and any release of memory must invoke *deallocate*. This approach, used throughout the ARTMOS design, allows substitution of different algorithms without impacting the rest of the operating system.

Pinkert and Wear (57:141–145) define a simple memory manager based on dynamic or variable block sizes. With such a scheme, memory is allocated in the amount requested. The algorithm described in (57) meets the requirements above that allocated memory be in one contiguous block and a task's memory cannot be moved once it has been allocated. Further, this algorithm recombines deallocated blocks to provide the largest possible avail-

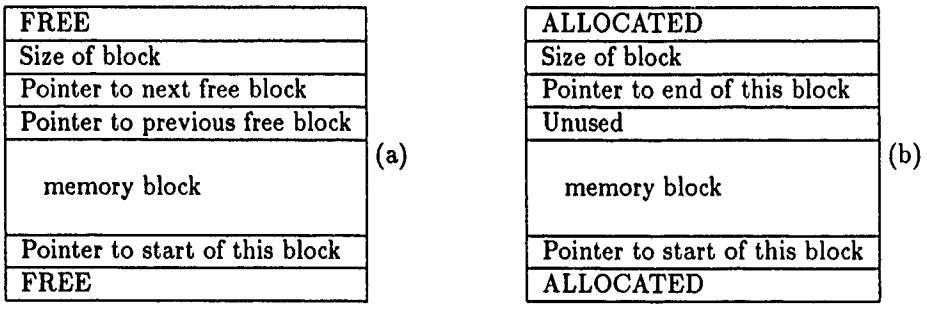


Figure 6.17. Structure of Blocks in Memory. (a) Available Block. (b) Allocated Block.

able blocks. Available blocks are managed in order of increasing memory address. The following description is abstracted from (57:141–145).

Memory is defined as a set of data structures, called memory blocks, containing six information fields, as shown in Figure 6.17 for an available and an allocated block. The information fields are used to allocate and deallocate the blocks. The next and previous pointers in the available block form a linked list of available memory. The FREE and ALLOCATED flags are used to simplify deallocation, allowing quick checks of the blocks immediately before and after the deallocated block. This permits adjacent available blocks to be combined into a larger contiguous available block. These flags also are used, with the the start and end pointers in allocated blocks, to insert the deallocated block in the available memory chain if adjacent blocks cannot be combined.

The memory allocation and deallocation operations can be greatly complicated by the need to continuously check for the beginning and ending of user memory when searching the available memory list. To avoid such checks, four dummy blocks are defined, two at the start and two at the end of memory. The initial memory list is shown in Figure 6.18.

Allocation of memory blocks could be implemented a number of different ways, the most common being first-fit, best-fit, and worst-fit (56:76). First-fit allocates the *first* block that is big enough; searching can start at the beginning of the list or where the previous search ended. Best-fit allocates the *smallest* available block that is big enough. Since the ARTMOS available memory list is ordered by memory address, the entire list must be searched. Worst-fit allocates the *largest* free block. Like best-fit, the entire list

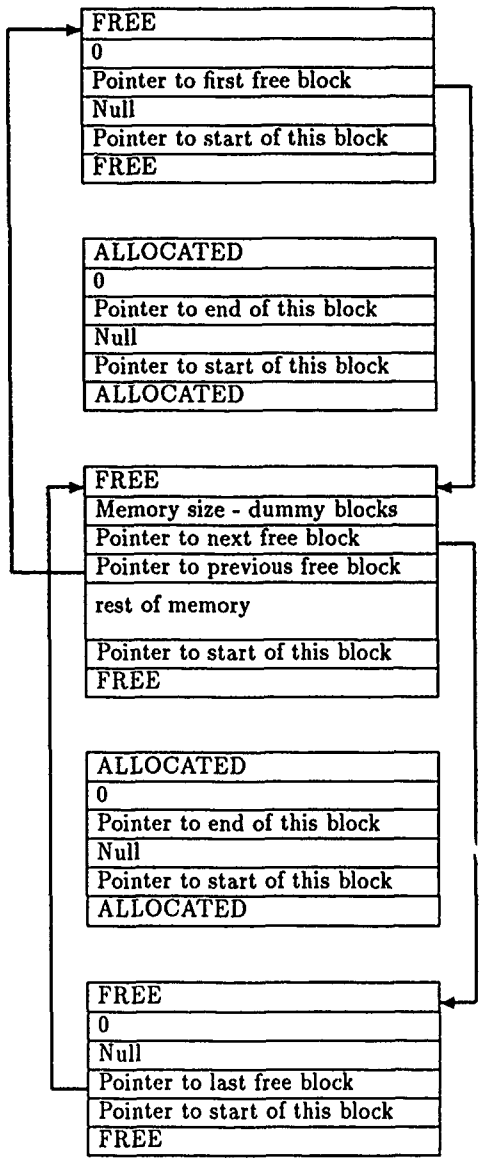


Figure 6.18. Memory List After Initialization

must be searched. The method selected by Pinkert and Wear, and used for the ARTMOS design, is based on the first-fit allocation technique, starting from the top of user memory for each search. If the selected block is sufficiently larger than the requested memory size, a new free block is created from the excess and inserted in the memory list. Otherwise, the entire free block is given to the requestor. Figure 6.19 shows the ARTMOS memory allocation algorithm.

Figure 6.20 shows the ARTMOS memory deallocation algorithm. When a memory block is deallocated, adjacent free blocks are collapsed, making the largest possible contiguous free areas available. This is accomplished by first checking whether there is a free block on either side of the returned block. If so, the returned block sand the adjacent free blocks are combined into a larger free block. When neither adjacent block is available, it must be inserted in the proper place in the available block list, which involves a search from the "middle" of the list (the returned block) to the fist previous available block and the first following available block. The returned block is linked between these two available blocks.

6.3.1.5 Interrupt Management The interrupt management requirements defined in Section 3.4.5 are straightforward and seem simple to implement:

1. The application can provide handler routines for specific interrupts,
2. the interrupt handlers return control to the operating system to allow task rescheduling, and
3. the application shall be able to enable or disable specific interrupts.

A new constraint is added to only permit an application to enable and disable interrupts that *belong* to the application, not those defined as belonging to the operating system.

These requirements, however, are more complex to implement then they first appear because of the need for applications to enable and disable specific interrupts. All processor architectures provide a mechanism associating each interrupt, exception, and trap with a unique *interrupt vector* or *interrupt number*. Most processors further provide an interrupt priority mechanism, allowing interrupts to be associated with priority levels so that the

```
ALLOCATE MEMORY (memory size, block pointer);
    FIND FREE BLOCK (memory size, block pointer);
    REMOVE SELECTED BLOCK (block pointer);
    IF ((block pointer.SIZE – memory size) > MIN BLOCK SIZE) THEN;
        MAKE NEW BLOCK (memory size, block pointer, new block pointer);
        INSERT FREE BLOCK (new block pointer);
    ENDIF;
END;
```

Figure 6.19. Memory Allocation

```
DEALLOCATE MEMORY (block pointer);
    next flag = FALSE;
    FIND NEXT BLOCK (block pointer, next pointer);
    IF (next pointer.STATUS = FREE) THEN
        COMBINE BLOCKS (block pointer, next pointer);
        next flag = TRUE;
    ENDIF;
    prev flag = FALSE;
    FIND PREVIOUS BLOCK (block pointer, previous pointer);
    IF (previous pointer.STATUS = FREE) THEN
        COMBINE BLOCKS (previous pointer, block pointer);
        prev flag = TRUE;
    ENDIF;
    IF (prev flag = FALSE) AND (next flag = FALSE) THEN
        INSERT FREE BLOCK (block pointer);
    ENDIF;
END;
```

Figure 6.20. Memory Deallocation

most important interrupts are serviced first. The Motorola 680X0 processor family, for example, has eight hardware priority levels (47). If a processor is servicing one interrupt, a new interrupt is recognized only if it has a priority higher than the interrupt currently being handled. Interrupts are masked by setting a flag in the processor status or flag register associated with a particular interrupt *priority* rather than a specific interrupt vector (57:8–9). This has the the undesired side effect of masking all interrupts of lower priorities, as well as any other interrupts associated with the same priority level. The only way to mask just one interrupt is to disable the interrupt at the hardware device.

Also, since the status or flag register used to mask an interrupt priority is part of the processor context that is saved and restored when tasks switch in and out of the processor. If a particular interrupt level were masked by one task, the processor status register is overwritten, and the interrupt potentially re-enabled, when the next task's context is restored. The interrupt would be disabled or enabled only when a particular task is running.

An interrupt mechanism is needed that allows specific interrupt vectors to be enabled or disabled, as well as providing some means of determining whether the interrupt "belongs" to the application or operating system. Further, the operating system should build a layer of abstraction on top of the hardware so that a common interrupt handling mechanism is provided, regardless of the target processor family (3:88). The interrupt mechanism should provide the primitives necessary for interrupt handling and protect the application developer from needing to know too much about the target. This is in keeping with one of the objectives of this thesis investigation, which is to minimize the impact of hardware specifics on the application by providing a programming interface that applies to a variety of target architectures.

Two components are used to implement the interrupt management mechanism described above. The first is the *Interrupt Table*, used to maintain information on each possible interrupt and indexed by the interrupt vector number. The second component is an interrupt manager routine invoked whenever an interrupt occurs. This approach is similar to that used in the Distributed Ada Real-Time Kernel (2:83–86),(3:86–93), though it has been tailored for the specific requirements of the ARTMOS.

	0	...	31	32	...	35	36	...	255
OWNER	reserved		reserved	OS		OS	Application		Application
CONDI-TION	bound		bound	bound		bound	unbound		unbound
STATE	enabled		enabled	enabled		enabled	disabled		disabled
PREEMPT	FALSE		FALSE	TRUE		TRUE	FALSE		FALSE
HANDLER	used by hardware		used by hardware	clock interrupt handler		receive interrupt handler	null handler		null handler

Figure 6.21. Interrupt Table After OS Initialization

The Interrupt Table is a fixed size structure containing an entry for each legal interrupt number. Each entry has five fields which contain information about the interrupt itself, whether it is reserved for use by the hardware and operating system, or used by the application, and information about the interrupt's handler. The OWNER field specifies whether an interrupt is owned by the operating system, the application, or reserved for hardware exceptions. The interrupt primitives apply only to interrupts that belong to the application. The CONDITION field indicates whether an interrupt handler has been identified for the specified interrupt. The PREEMPT flag indicates whether the interrupt can caused the interrupted task to be preempted, and is set by the application when a handler is bound to the interrupt. STATE is the state of the interrupt, either enabled or disabled. All interrupts are initially disabled. Finally, the address of the interrupt handler is stored in the HANDLER field. Figure 6.21 illustrates the structure of the Interrupt Table.

The second aspect of interrupt management is the interrupt manager, shown in Figure 6.22. When an interrupt occurs, PROCESS INTERRUPTS is invoked. The Interrupt Table entry associated with the interrupt is check to see if the interrupt is enabled. If the

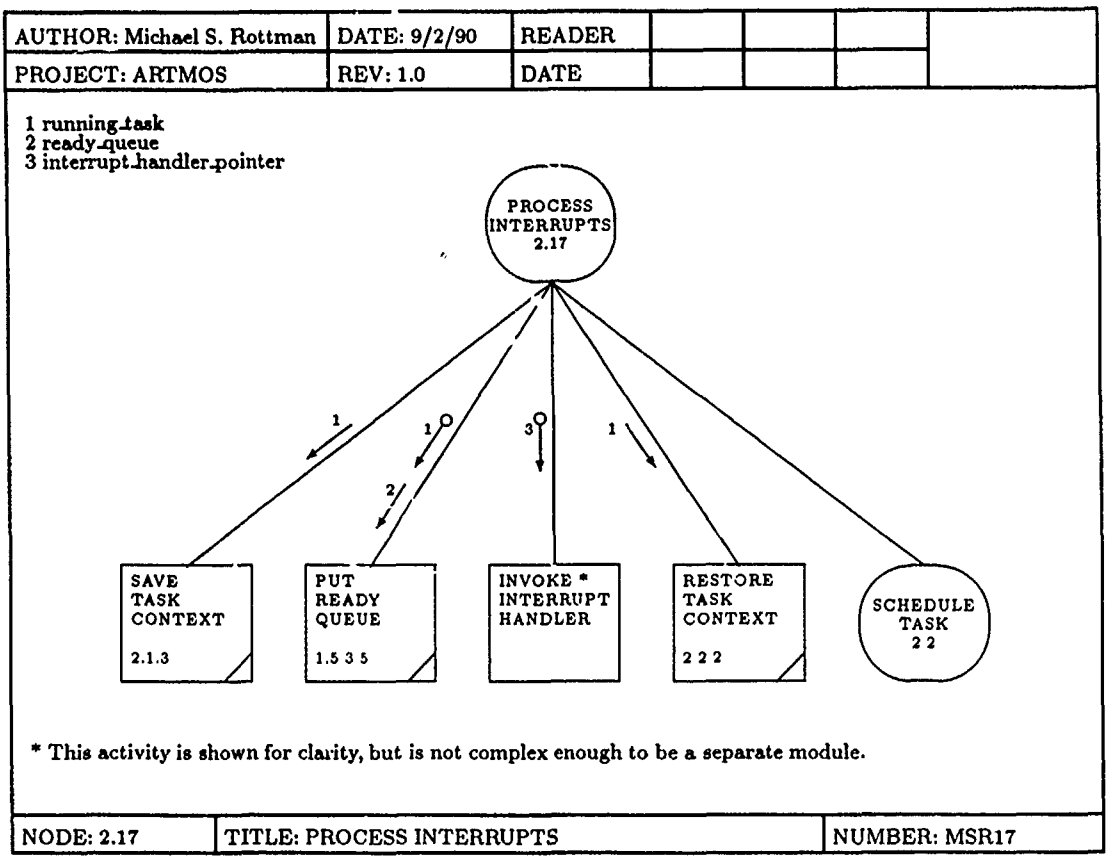


Figure 6.22. Design for the Interrupt Manager

interrupt is disabled, it is dismissed and the task is resumed with minimal delay. If the interrupt is enabled, the context of the interrupted task is saved and the interrupt handler identified in the Interrupt Table is called. After the interrupt is serviced, one of two things happens, depending on whether the interrupt is flagged as preemptive or non-preemptive in the Interrupt Table. Non-preemptive interrupts cause the interrupted task to be resumed after its context is restored, while preemptive interrupts caused the scheduler to be invoked.

Three system call primitives provided by ARTMOS allow the application to perform interrupt management: BIND INTERRUPT HANDLER, ENABLE INTERRUPT, DISABLE INTERRUPT. The application uses the BIND INTERRUPT HANDLER call to identify an interrupt service routine that is called when the indicated interrupt occurs. This call also specifies whether or not the interrupt is preemptive. DISABLE INTERRUPT

changes the state of the indicated interrupt to *disabled*, causing the operating system to ignore all occurrences of that interrupt by not calling its associated handler. ENABLE INTERRUPT both initially enables an interrupt and re-enables the interrupt after it has been disabled by setting the interrupt state to *enabled*. When a specified interrupt is enabled and the interrupt occurs, the bound interrupt handler is called.

The operating system performs two additional interrupt management functions not available to the application: masking and unmasking of the processor interrupt priority levels. System calls must be handled as quickly as possible to minimize the operating system overhead when the application is not running. Even more important in real-time systems, the execution time of operating system functions must be predictable and be bounded in the worst case. If interrupts are allowed during operating system calls, the amount of time to handle the call is unpredictable. Also, many system calls manipulate operating system data structures, such as the ready queue or memory list. Interrupts must be masked while altering these data structures to prevent concurrent, unprotected access to the same data. Care is taken to minimize the amount of time when interrupts are masked, however, to reduce the possibility of missed deadlines.

6.3.1.6 Semaphore Management The ARTMOS provides support for binary semaphores. As specified in Section 3.4.3, semaphores are visible only on the processor on which they are declared, and so can be used only by tasks local to that processor to protect access to some local resource. Tasks can specify a timeout value when attempting to claim the semaphore, thus preventing tasks from blocking indefinitely for a resource. The application must determine how long tasks can afford to wait for semaphores, and what to do if the resource cannot be claimed within that time frame.

As described by (3, 48, 57, 59), tasks that are blocked waiting for a semaphore to become available are kept on a queue structure associated with that semaphore. When a task releases the semaphore, a task in the semaphore queue is unblocked and granted the semaphore. If the queue is empty, the semaphore becomes available. The semaphore data structure is therefore must consist of of a flag, which may be either *available* or *claimed*, and a queue structure comprised of a count of the number of tasks in the queue and

pointers to the front and rear of the queue. The application uses this data type to declare all application semaphores, which initializes the flag to *available*, the count to zero, and the pointers to null.

Two system calls are provided to allow tasks to request and release semaphores: WAIT SEMAPHORE, and SIGNAL SEMAPHORE. WAIT SEMAPHORE attempts to claim the semaphore for the calling task. If the semaphore is available, the flag is changed to *claimed* and the calling task resumes execution. If a task already has the semaphore, the calling task is added to the queue of tasks waiting on that semaphore. The SIGNAL SEMAPHORE call releases the semaphore from the calling task. If the queue is nonempty, a task is removed from the queue and given the semaphore; otherwise, the semaphore flag becomes *available*.

The second requirement above, that wait time for semaphores be bounded, adds significant overhead to the ARTMOS semaphore management facilities. To implement timeouts, tasks blocked on semaphores queues must be checked every time the processor time variable is incremented or synchronized to determine whether any should unblock at the new time. The application declares semaphores to the operating system with the BIND SEMAPHORE system call, so that the operating system is aware of all semaphores. The semaphore management overhead stems from the need to search each semaphore queue after each clock change and the manner in which the queues are ordered.

The semaphores queues can be ordered by the priority of the tasks, by timeout values, or by order of blocking (FIFO). Many real-time operating systems, such as DARK (3:94), implement FIFO semaphore queues: blocked tasks are added to the rear of the semaphore queue, while tasks are removed from the front. This approach has two deficiencies. First, higher priority tasks could get stuck behind lower priority tasks in the queue, causing more important deadlines to be missed in favor of less important time constraints. Second, the entire queue must be searched whenever the clock changes for tasks whose timeout has expired since there is no correlation between queue ordering and timeout value. This second problem could be addressed by ordering the semaphore queue by the timeout value of the blocked tasks. The overhead of checking for and waking up tasks with expired deadlines is greatly reduced because tasks with the nearest timeout are at the front of the

queue. However, lower priority tasks can still prevent higher priority tasks from getting the resource, if they have nearer deadlines. The third approach is to order the queue by the priority of the blocked tasks. The advantage to this approach is that the most important tasks claim resources first. If deadlines *have* to be missed, they will be the less critical ones. Like the FIFO implementation, however, the priority queues must be searched to find tasks whose timeout has expired.

The approach selected for the ARTMOS detailed design is to order the semaphore queues by task priority. In real-time systems, it is critical that the most important tasks get the resources as soon as possible, so the most important deadlines can be met. This is consistent with the use of a prioritized ready queue and preemptive scheduling: the most important ready task should be executing at all times. The FIFO and timeout-ordered semaphore queue approaches are not suitable because they would allow higher priority tasks to wait while a lower priority task claims the semaphore. As discussed above, however, use of prioritized semaphore queues requires that the operating system search all queues for tasks whose timeout expired when the clock changed.

Figure 6.23 shows the algorithm of the primary routine used to check for and wake up tasks with expired timeouts, WAKE SEMAPHORE QUEUE. This routine is invoked during the servicing of the two interrupt handlers which alter the time value, CLOCK INTERRUPT HANDLER and SYNCH INTERRUPT HANDLER. The handlers call WAKE BLOCKED TASKS to check all of the semaphore queues. WAKE BLOCKED TASKS in turn invokes WAKE SEMAPHORE QUEUE for each semaphore declared by the application. WAKE SEMAPHORE QUEUE uses the brute force approach: starting at the front of the semaphore queue, it checks each task waiting in the queue. If a task's timeout has expired, it is removed from the semaphore queue and put in the ready queue. More sophisticated methods to perform the same function may be possible, but they are outside of the scope of this thesis investigation.

It should be noted that the implementation of the semaphore queue is insulated from the rest of the system. All details pertaining to the specific implementation are hidden in the three routines which manipulate the queues: GET SEMAPHORE QUEUE, PUT SEMAPHORE QUEUE, and WAKE SEMAPHORE QUEUE. If different queue imple-

```
WAKE SEMAPHORE QUEUE (sem pointer);
    POINT FRONT SEMAPHORE QUEUE (sem pointer, task pointer);
    WHILE (task pointer ≠ NULL) DO
        READ TIMEOUT (task pointer, timeout);
        IF (timeout ≤ time) THEN
            pointer = task pointer;
            task pointer = task pointer.NEXT;
            REMOVE FROM QUEUE (sem pointer, pointer);
            PUT READY QUEUE (pointer);
        ELSE
            task pointer = task pointer.NEXT;
        ENDIF;
    ENDDO;
END;
```

Figure 6.23. Wakeup of Blocked Tasks in a Semaphore Queue

mentations are needed for different applications, the only changes need to be made to the above routines; the rest of the system is not affected.

6.3.1.7 Clock Synchronization Section 3.4.6 described the need for keeping the local clock values on the processor nodes synchronized within some predefined window. If the local clocks are permitted to skew with respect to one another over time, the timing of the task interactions can become unpredictable, though the communications protocol still protects the integrity of the data exchanges. A slow node clock could prevent tasks on that node from producing data when the rest of the system expected it, or could result in data produced on other nodes from being consumed when needed.

There are many different methods, both in hardware and software, to synchronizing distributed clocks. The iPSC/2 hypercube architecture provides no hardware support for clock synchronization, so the ARTMOS must perform all synchronization in software. The most obvious approach is to select one of the node clocks as the standard, as done in DARK (3:99–100) and the Fault Tolerant Multiprocessor (FTMP) (12:194), but this assumes that the selected clock is reliable. A more reliable approach is to periodically compute a clock correction factor on each node (12:194). One such method is for each node to broadcast its local clock value to the other nodes at periodic intervals. Upon receiving a clock value

from another node, a node computes the difference between the received clock value and the local clock value, storing the result in a table. Over time, each node accumulates a history of clock error values which can be evaluated to compute an "average error" value with which to adjust the local clock.

The problem with implementing this approach in the iPSC/2 (or any other hypercube architecture) is that the clock values must be broadcast to all nodes. For 2^N processors, a broadcast requires N transmit steps, with significant operating system overhead to route the clock values at the intermediate nodes. Such an approach would also increase the complexity of the ARTMOS communications software. To reduce this overhead, a variation of this approach was selected in which each interprocessor datagram contains the node time at which the datagram was transmitted. On receiving nodes, the received times are used to compute clock error histories, as described above. Each node thereby computes its clock correction based on a *subset* of all possible clock error values, significantly reducing the overhead associated with transmitting special clock messages. This algorithm assumes that the nodes communicate with most of the other nodes in the system rather than just one or two, which could reduce the accuracy of the correction factor due to the lack clock data from the entire system.

Two routines and two global variables implement this clock algorithm. COMPUTE CLOCK ERROR is called by the receive interrupt handler to calculate the difference between the source node time when the message was sent and the local time. The difference is added to total error, a global sum of the computed errors, and number error values is incremented. The SYNCH CLOCKS routine computes the average clock error from these two variables and adjusts the local time by the result. The variables are then reset to zero.

Many variations are possible on these approaches, such as discarding the highest and lowest error values, or computing the mean instead of the average. Statistical analysis of clock correction during operation may also provide for minute adjustments to the programming of the clock device to correct for skewing. Research into more advanced or sophisticated clock synchronization algorithms is beyond the scope of this thesis investigation. The ARTMOS design, however, is such that a change to the clock synchronization algorithm requires only changes to the SYNCH CLOCKS routine and the routine in RE-

CEIVE INTERRUPT HANDLER that processes received clock values. The rest of the ARTMOS is unaffected.

6.3.2 State Transition Manager The second step in the detailed design phase of the design methodology outlined in Section 5.2.1 is to determine whether a state transition manager (STM) is needed for the ARTMOS, and to design it if necessary.

The DARTS design methodology uses structured design (25:942) to decompose each concurrent process into its lower level components and interfaces. Structure design aplies transform or transaction centered analysis to derive structure charts from the data flow diagram model of the system. A major limitation to transaction analysis occurs when the real-time system is state-dependent, where actions taken on incoming data depend not just on the data but on the current state of the system (25:942).

For such state-dependent real-time systems, DARTS uses state transition managers to provide an abstraction encompassing all state conditions and transitions. The STM is a module that maintains the current system state and a state transition table defining all legal and illegal states. Tasks needing to process a transaction call the STM with the desired action as an input parameter. The STM checks the tables to determine if the desired transaction is legal for the current system state. If the transaction is legal, the STM changes the state (if necessary) and returns a positive response to the calling task. Otherwise, a negative response is returned. The state transition table data structure is hidden from the calling tasks, whose only access to the STM is through defined access procedures to check the validity of task requests and perform the state transitions. LVM/OOD defines a similar approach using finite state machines (51:72).

The application of structured design to the ARTMOS made extensive use of transfrom analysis to generate structure charts from the initial SADT diagrams. Tranaction centered analysis was not necessary because the ARTMOS is not a state-dependent system. Many of the system calls available to the application tasks require that the task be in the *running* state, but this does not require a STM. Since only a running task is capable of requesting a service, there is no need to validate the task's request. Applications designed for the ARTMOS may require a STM, but the ARTMOS itself does not.

6.3.3 Encapsulate into Modules The third and final step of the detailed design methodology defined in Section 5.2.1 partitions or encapsulates the ARTMOS software routines and operations into logically related groups or "modules."

This design step is a variation of one performed in the LVM/OOD methodology (51:212). In LVM/OOD, the independent processes (defined as Ada tasks in that methodology) are grouped together into Ada packages because tasks cannot exist as independent compilation units. Like other Ada constructs, the package is comprised of a *specification* and a *body*. The specification contains information exported to users of the package, such as data structures or subprogram definitions. Package users are restricted to manipulating the declared data objects with the declared operations. The actual implementation of the operations is in the package body, inaccessible to the user. This approach promotes information hiding, since the specification exports the minimum information needed to use the package while hiding the implementation details within the body. Further, the implementation of the body can be changed without requiring modification to any users of the package, so long as the package specification is not changed.

The advantages of this approach are apparent, but Ada is the only language providing explicit packaging constructs. When Ada is not the implementation langauge, however, packaging can be replaced with appropriate collections of "include" files (51:222), called modules here. This is the approach used in the ARTMOS design. These files could be separately compiled and linked, or "included" into a master module for compilation.

The goal of this design step is to perform a preliminary partitioning of the ARTMOS design into logically related modules or "packages." These modules could be implemented as separate files. This partitioning is intended to provide a modular, flexible implementation derived from the operations and abstractions defined during the design process.

Nielsen and Schumate (51:113–115) provide a set of nine heuristics for creating application packages. Most of these heuristics are relevant to the ARTMOS design, but several require modification or deletion since the ARTMOS does not presuppose Ada as the implementation language. The modified heuristics applied to partitioning the ARTMOS design into modules are:

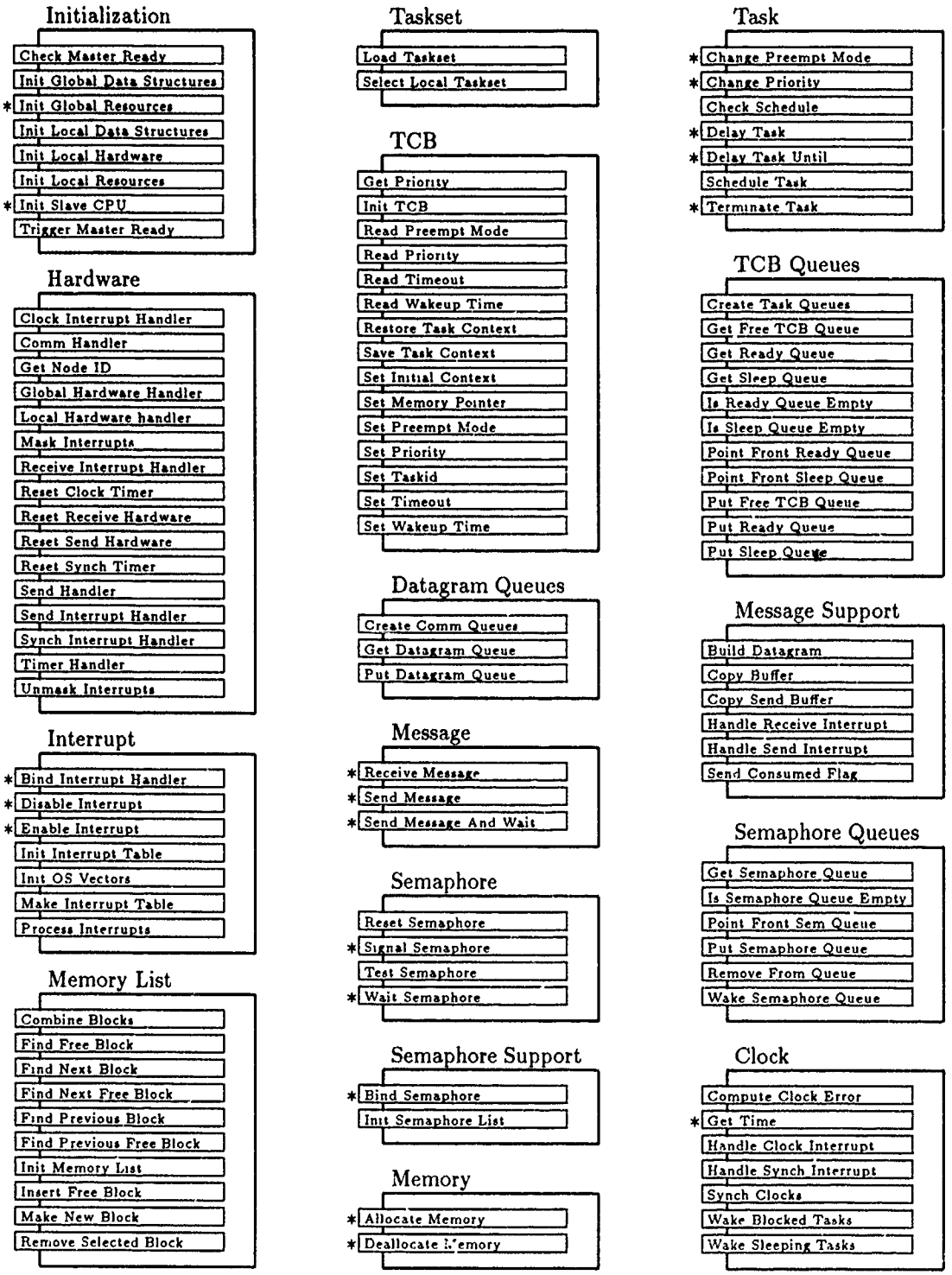
1. Group routines within a module if the routines are functionally related.
2. Special purpose routines that are generally applicable or serve as library units can be combined into a module.
3. Attempt to construct modules representing objects.
4. Group routines to minimize coupling with respect to data types, operations, and constants imported from other modules.
5. Combining of routines into modules should result in minimum dependency between modules.
6. Combining of routines into modules should result in minimum recompilation when any module is changed.

Essentially, the partitioning decisions are made to maximize modularity and localization of data structures and their associated operations. This makes the software easier to implement, test, and maintain.

Figure 6.24 shows the ARTMOS module encapsulation. A variant of Booch's package symbol (9:219) is used here to depict an ARTMOS module. The module is shown as a rectangle with "windows" on the left side to show the operations provided by that module.

Many of the modules shown in Figure 6.24 are derived from the application of object-oriented design methods during the ARTMOS detailed design phase. Whenever possible, specific operations were defined to act on specific data structures, thereby controlling access to the data object while hiding details of the object's implementation from the user of the object. Examples of object manager modules are the Memory, Task, and TCB modules. Without needing to know how memory is defined or managed, other routines can obtain and release memory by using the ALLOCATE MEMORY and DEALLOCATE MEMORY operations provided by the Memory module. The Task and TCB modules are also object-oriented, but they also demonstrate the concept of a "sub-object." Operations on the task object utilize the operations on the TCB object. The advantage of this approach is that if the TCB structures changes, only those routines in the TCB module require changes; the task operations are unchanged.

Other of the modules are comprised of routines which perform similar or related functions. Examples of this type are the Intialization, Semaphore Support, and Message Support module. An additional module, Hardware, consists of all routines and handlers



* denotes system call handlers

Figure 6.24. ARTMOS Module Partitioning

that deal directly with the local processor and communications hardware. The ARTMOS design specifically included these hardware interface routines to minimize the impact of different target computer architectures on the software. Most of the ARTMOS does not interact with the hardware at all; the parts that must interact with the hardware do so through the operations defined in the Hardware module.

6.4 Chapter Summary

This chapter discusses the detailed design of ARTMOS presented in Appendices B, C, and D. The chapter begins by giving further consideration to the design methodology. The application of SADT diagrams to structured design and the overall design method is examined, and extensions are made to the basic structure chart notation to increase its suitability to the ARTMOS design. The chapter then selects a target computer architecture, the Intel iPSC/2 hypercube machine, for the ARTMOS implementation, and briefly describes the architecture. Next, the chapter describes in detail the decomposition and low-level design of the independent processes identified in the preliminary design and the partitioning of the resulting operations and data structures into logical "modules," similar to Ada packages. The discussion emphasizes specific decisions made and alternatives considered at each stage of the detailed design.

This discussion of the detailed design concludes the description of the ARTMOS development phase. The next chapter examines the final ARTMOS design with respect to the original objectives of this thesis.

VII. Analysis and Recommendations

This chapter presents an analysis of this research with respect to the specific objectives defined in Chapter 1. The first section is an analysis of the specification and design of the ARTMOS. A discussion on the testing of the ARTMOS is also included. Section 2 examines the mapping of the ARTMOS design to a target parallel architecture and evaluates several aspects of the detailed design. The final section is an analysis of the viability of the design for real-time applications and the applicability of the application interface to other target architectures.

7.1 Analysis of ARTMOS Specification and Design

One objective of this research was to specify and design a real-time multiprocessor operating system that incorporates the AMCAD application program interface (API). This section analyzes the results of this objective for those aspects of the specification and design that are independent of the selected target architecture. Specific areas examined are the impact of emphasizing software engineering in the ARTMOS development, the use of shared variables, and the testing of the ARTMOS design.

7.1.1 Software Engineering Impact The software engineering principles of abstraction, information hiding, and modularity are extensively applied to the ARTMOS specification and design in accordance with Section 3.6.1 of the requirements document. These principles are explicitly factored into all decisions in each phase of the development process. The detailed design phase also emphasizes the use of object-oriented design in the ARTMOS decomposition as appropriate as described in Section 5.2.2.

The benefits of these software engineering principles are well documented. In general, effective application of software engineering methods increases the understandability, reliability, efficiency, and maintainability of the designed software (9:25). Object-orientation provides a means of structuring data and related operations to enhance the functionality and reliability of the software (92:104). Many of these benefits are apparent in the ARTMOS design.

The ARTMOS incorporates several different levels of abstraction. An example of this abstraction is the ARTMOS task management section. At the highest level, operations are provided to select and load *tasksets*, which are collections of tasks for execution on given processors. The next level of abstraction manipulates individual *tasks*, providing means for them to be delayed, terminated, scheduled, and their characteristics altered. Tasks are represented at yet a lower level by *task control blocks*. To implement the higher level task abstraction, functions are provided to move TCB objects between task management queues. The lowest level of abstraction manipulates the internal components of the TCB. Other examples of abstraction in the ARTMOS design are the memory management, semaphore, and communications sections.

Complementing the levels of abstraction designed in the ARTMOS is the principle of information hiding. Only information needed to use a particular level of abstraction is provided to the user of the level; all other information is "hidden." The user is told how to use the provided functions, but not how the functions are implemented or the nature of the underlying data structures. The TCB queue management section, for example, contains routines that add and remove TCB objects from the various TCB queues. Knowing how the queues are implemented or how the TCBs are structured is not essential to using the routines, so it is hidden from the routine's users. All the user needs to know is what operations are available on the TCB queues and how to use them. Similarly, memory can be acquired or released by calling the appropriate routines, all without knowing or caring about the underlying memory management algorithms and data structures.

A modular design partitions the software into distinct modules, each of which implements a specific function or collection of related functions. Loosely coupled modules require less information to understand one another, allowing them to be coded, tested, and maintained relatively independently. If the elements of a module are closely related, the module is highly cohesive and more understandable, reusable, and maintainable. In the ARTMOS design, modularity is apparent in both the low level functional decomposition and the more global "file modules" defined in Section 6.3.3. Though the routines defined in the functional decomposition are not necessarily "modules" per se, they are defined to be "modular." Each routine performs a distinct function, independent from other routines

to the extent possible. The information required to use or interface to routines is clearly defined. Likewise, the file modules bundle together routines that perform similar functions or which operate on the the same data structure. Interfaces between file modules are well defined and consist of only the information needed to use the routines or data structure. The implementation of the routines is hidden from the user of the module.

Object-oriented programming, "when understood and properly used, makes software development faster, easier, and more reliable than procedural programming. It also helps you achieve the goal of designing software that you can understand, modify, and reuse" (11:265). An object is a software entity comprised of a data structure and the operations for manipulating the data structure. How the data is represented and the details of the operations that act on the structure are hidden from all code outside the object (11:26). The concept of object-orientation is informally used in the ARTMOS design. Many of the file modules described above are organized as objects. The TCB module, for example, defines the actual TCB data structure and all possible operations on the TCB structure. None of the other modules need to know the exact structure of the TCB, nor the details of the manipulation routines. Similar "object" modules are defined for Messages, Memory, Semaphores, Tasks, and so on. The ARTMOS object-orientation does not support such advanced object properties as inheritance and polymorphism, which result from the arrangement of different kinds of objects into a class hierarchy (11:266). Inheritance means that an object has access to (inherits) the operations and data of all of its superclasses in addition to those of its own class. Polymorphism refers to an object's ability to make the same request of different objects and achieve different results from each.

The emphasis on abstraction, information hiding, modularity, and object-orientation in the ARTMOS design has led to several design characteristics that directly support the objectives of this thesis effort. These characteristics are examined next.

7.1.1.1 Portability One driver of this research is portability, both of application software and the operating system itself. Portability of application software is achieved by providing a standard application program interface (API) with the operating system that can be consistently implemented across a variety of architectures. If the pro-

gramming model is not violated in implementations of the operating system for different targets, applications software can be moved between targets without recoding (because of the operating system).

Operating system (OS) portability is more complicated because of the many potential target architectures and hardware implementations. True OS portability (simply recompiling the OS for each new target) is impractical, if not impossible: at some point, the OS code needs to be modified to accommodate the hardware and architecture differences of the new machine. A more practical approach is to design the OS so as to reduce the amount of code that must be changed to port to a new target. Section 7.2.1 describes how this is accomplished in the ARTMOS design.

7.1.1.2 Modifiability One benefit of the emphasis on object-orientation and modularity is that the ARTMOS design is extremely modifiable. Similar functions or routines that operate on the same data structure are grouped together into modules with clearly defined functions and interfaces. The definition of objects where appropriate groups data structures and their associated operations into distinct modules. This partitioning, both functional and object-oriented, serves to limit the impact of making changes to the software. Modification to a data structure may require changes to the routines that act directly on that data structure, but should not affect users of those routines. With this approach, the underlying object may change without recompiling or modifying the code that uses the object. In the ARTMOS design, for example, a change in the memory list implementation may force changes to the routines that manipulate the list. These changes are transparent to the higher-level memory allocation and deallocation operations.

Certainly a primary motivation for this emphasis is that "we don't know all the answers. A real-time operating system that is state-of-the-art today will be obsolete and inadequate tomorrow unless it is designed to grow and change" (92:104). A modular, object-oriented OS provides that capability. Designers can add a hard deadline scheduler or a rate monotonic scheduler, a new intertask communications algorithm, a new memory management paradigm, or whatever is required for a given application with minimum disruption to the existing operating system and applications software.

A further advantage of the modular ARTMOS design is that it provides a flexible testbed capability for the investigation of real-time operating system and parallel software development issues. The ARTMOS design emphasizes the definition of a modular OS framework that implements the specified application interface. Lower-level routines can be easily modified to implement new algorithms or compare alternative approaches. The ARTMOS can provide AFIT students with a tool for learning about real-time operating systems as well as experimenting with parallel algorithms and implementations.

7.1.1.3 Testability Another consequence of modularity and object-orientation is increased testability of the ARTMOS implementation. The modules are defined to be highly cohesive, with minimal coupling between modules. Since fewer inter-module dependencies exist to complicate testing and implementation decisions are encapsulated within the modules, each module can be tested separately by simulating the other modules with which it interacts. Testing is discussed in greater detail in Section 7.1.4.

The design is clearly partitioned into objects and functional modules, easing testing after the software is modified. Because the implementation of the functions or data objects in a module are not visible to code using the module, changes to the underlying implementation force re-testing of that module but not of the using code. This is a tremendous advantage over non-modular design, where the entire software most be re-tested after each change, no matter how trivial.

7.1.1.4 Maintainability Finally, the emphasis on object-orientation, information hiding, and modularity has produced a highly maintainable design. All major data structures are defined as objects, with associated operations on the structures provided with the object. Access to the data objects is limited to the provided operations. Related functions not associated with particular data objects are collected into functional modules. Each entity (module and object) is well documented with clearly defined interfaces and functionality.

The resulting design is highly maintainable. Changes to a modules are limited to that module, reducing the retesting required after each change. Since related functions

or functions acting on the same data object are kept in one module, routines can be modified without having to trace through the entire program to make sure all other routines impacted by the change are also modified. The well-defined functionality of the design simplifies the process of identifying which modules need modified and determining how the modules interact with the rest of the system.

7.1.2 Use of Shared Variables Section 5.3.5.1 identifies the need for a limited number of shared variables in the ARTMOS design, such as the ready task and sleeping task queues and a pointer to the currently running task. The term "shared variable" does not refer to variables shared by the multiple processors, as is typically the case. In the ARTMOS, each processor has a unique copy of the operating system software and system data structures which are used to manage the tasks assigned to that processor. ARTMOS shared variables are globally accessible to all local operating system routines, but not to the operating system copies on other processors.

Many of the data structures originally envisioned as needing to be shared variables never needed to be used as such in the design. The ready queue, for example, is never accessed directly by the higher-level operating system operations because queue manipulation routines are provided. All access to and manipulation of the ready queue occurs through the queue handling routines. Likewise, the memory list, interrupt table, and semaphore list are never accessed directly. Specific routines are provided to allow other routines to use the data structures. In each of the above cases, the data structure is treated as an *object* rather than a shared variable: the data structure is defined and specific operations are provided which act on that structure.

Several variables *are* shared in the ARTMOS design, however, the most notable being *running task*, the pointer to the currently executing task. Only two operations directly access running task, those being SAVE TASK CONTEXT and RESTORE TASK CONTEXT. Many of the system call handlers must access the shared pointer in order to pass it as a parameter to lower-level routines. The routines use running task as they would any other task pointer. Another frequently accessed shared variable is the local clock variable *time*. In each case, the shared variable is *read* by multiple routines but

can only be *written* by one specific routine. SCHEDULE TASK is the only routine that changes running task, while the HANDLE CLOCK INTERRUPT routine sets the clock variable.

7.1.3 General Purpose Routines All of the low-level support operations, such as the queue manipulation and TCB manipulation routines, accept as a parameter a pointer to the task on which they operate. In many cases, ARTMOS only uses the routines to operate on the running task, so the parameter would seem to be extraneous. The driving factor in passing this parameter, however, was to provide a flexible, expandable, and maintainable design. Later additions or changes to the ARTMOS may require that the routines operate on any task, not just the running task. These routines should not have to be redesigned to give them more general purpose.

Some routines operate on specific data structures by definition. The ready queue manipulation routines act only on the ready, not the sleep queue. This is necessary since the ordering of the two queues is different and the same routines cannot work for both schemes. These routines are special purpose. With the task and TCB manipulation routines, on the other hand, the operations are identical regardless of which task is being manipulated. These routines are considered general purpose. Wherever possible, the ARTMOS design uses general purpose routines.

7.1.4 Testing ARTMOS The primary objective of any software development effort is to correctly implement the requirements specification. Unfortunately, each step of the development process is subject to error or misinterpretation. According to (50:103-104), "software development is largely a process of communicating information about the eventual program and translating this information from one form to another ...the vast majority of software errors are attributed to breakdowns, mistakes, and noise during the communication and translation of information." To make matters worse, the cost of pinpointing and correcting errors increases dramatically the later in the development cycle they are detected. "The warranty and support costs of fixing an error that's gotten into the field are 35 times greater than correcting it at the design stage" (13:4).

Clearly, it is necessary to increase the quality of the each step of the development process by detecting errors earlier, while the cost to remove them is relatively low. Some means is needed of testing or verifying each development phase to evaluate its correctness and consistency with the preceding phase. Two approaches to evaluating development correctness are formal proofs and testing.

7.1.4.1 Formal Proof of Correctness One means of verifying that no errors exist is by formal mathematical proof of program correctness (32, 33, 50). The most common method of program proving is *inductive assertions*, in which a set of formal theorems, or assertions, is developed about the program based on first-order predicate calculus. Formal proof of the theorem set guarantees the absence of errors in the program. Other proof techniques include methods of predicate transformers, subgoal induction, computation induction, structural induction, and intermittent assertions (50:152). Another approach is UNITY, which focuses on a model of computation based on unbounded nondeterministic iterative transforms of the program state, a notation for specifications, and a theory for proving the correctness of the specifications (15:18). UNITY metaprogram development consists of iteratively refining the program specification and proof from a general solution strategy to the final specification from which code is derived (15:7).

The program proving approach is not without problems, however. The techniques are "extensive, costly, manual, and generally require much time and experience" (32:470). As a result, no large programs have been formally proven correct (50:153). Large programs could be partitioned into smaller, less complex pieces (modules) that could be individually proven, but the effort to accomplish this is still considerable. Also, there are studies showing errors in published proofs, showing how proofs can be fallible and why "proven" programs still need to be tested (50:153).

A further consideration is that formal proofs show the correctness of a program *implementation*; they do not address the requirements specification and design phases. A program could be a logically correct implementation of incorrect specifications, or not correspond to the design specification. Testing is still required to examine how well the program meets the requirements and design specifications.

7.1.4.2 Testing for Correctness The second approach to evaluating software correctness is testing. "Testing and evaluation of all phases of the software life cycle must be an integral part of any software engineering approach" (32:467). The intent of testing is to *find* errors: a "successful" test is one that detects an error, and an "unsuccessful" test is one that causes a program to produce the correct result (50:6). From this definition, a test that produces the expected output does not prove that no errors exist, it merely signifies that no errors were *found*. Testing "can demonstrate the presence of errors, not their absence" (50:467). Though it may be impossible to prove through testing that a program is error-free, it *can* provide confidence that none exist.

Life cycle testing of a software development can be achieved through static or "human" testing and dynamic "execution" testing.

7.1.4.2.1 Static Testing Static testing of a software system applies test and analysis methods that do not require actual execution of the software (33:102). Detailed analysis, inspections, walkthroughs, and simulated execution are used to evaluate the extensive documentation produced by the requirements, design, and implementation phases. The objective of this testing is to test the correctness, completeness, consistency, and clarity of requirements, the design structure, and the coded program (32:467). The result is a higher quality software program.

Static testing of the requirements is performed to ensure that the software requirements are completely and correctly defined (24:117). This is accomplished by inspection and analysis of the requirements specification for consistency, necessity, sufficiency, feasibility, and testability (33:104). Each software requirement must be consistent with other requirements and traceable back to the system requirements. Each must be correctly represented. The entire set of requirements must be evaluated for completeness and proper allocation of requirements to software functions (24:117). Requirements must be testable or they serve no purpose.

Design analysis is performed to ensure that the design is correct and satisfies the requirements specification (24:18). Two methods employed to test the design are inspection and the structured walkthrough, or simulated execution, which traces program execution

with representative inputs. The goal of design analysis, as with requirements analysis, is to evaluate the design for consistency, correctness, necessity, and sufficiency (33:105). Completeness is tested by correlating design elements to the source requirements to ensure that all needed and no extra functionality is included. Techniques are applied to evaluate the correctness of the mathematical equations, algorithms, and control logic. All aspects of the interaction between modules must be completely specified and visible, with only necessary data passed between modules. Modules must be used consistently wherever they appear in the design (24:119).

Program analysis verifies that the coded software is a correct implementation of the specified design (24:119). Module routines are checked for adherence to specified standards, correct symbolic execution, correct expression evaluation, correct data types, and correct routine parameter passing (32:468). Code analysis also can find poor programming practices.

These analysis steps are intended to improve the quality of the software by testing and reviewing each phase of the development to produce more consistent and correct requirements and design specifications. The ARTMOS has undergone the requirements and design analysis described here, primarily through inspection and structured walkthroughs. The result is a high degree of confidence that the detailed design correctly reflects the functional requirements. Program analysis should be applied when the design is actually implemented.

7.1.4.2.2 Dynamic Testing Dynamic testing searches for errors by executing portions of a program with representative valid and invalid inputs followed by analysis of the outputs (50:5-7). Dynamic testing can be either *black-box* or *white-box*. In black-box testing, each software unit, sub-system, or system is treated as a "black box" that does its work invisibly (13:6). Test data is selected to check whether the code performs as specified for valid, invalid, boundary, and random inputs. Types of black-box testing include equivalence class partitioning, boundary value analysis (13:6), and exhaustive input testing (50:8). Black-box testing is intended to show that code meets its requirements, and so is unable to detect inefficient code, unspecified functionality, or unused code (13:7).

White-box testing examines the internal structure and logic of the software (50:9). Common types of white-box tests include code walkthroughs (13:7) and path coverage tests (50:9), which seeks to execute with test data all paths of control flow through the software. White-box testing can find unused or inefficient code that black-box testing might not, but cannot find missing paths and might miss data-sensitive errors. Automated tools can greatly reduce the overhead of black-box and white-box testing by test set generation, code and path coverage analysis, and critical path testing (13:8–9).

Exhaustive testing of all valid and invalid inputs and all possible paths in a program is impossible (32, 50). Judicious selection of specific tests, however, can result in a high level of confidence in software quality (32:470).

To achieve some degree of control over the process of locating, analyzing, and correcting errors, testing begins at a low level and proceeds to more complex software elements. Based on this strategy, the test process is typically broken into distinct phases: module testing, integration testing, and system testing (13, 50).

Module testing is a process of testing large programs in small manageable units, such as modules, subprograms, or subroutines (50:77). This approach eases the process of locating and correcting errors, since any errors can be directly attributed to the tested module. Each module is individually black-box tested by supplying it with inputs from dummy "driver" modules and providing dummy "stub" modules to respond to any calls made by the test module (32, 50, 58). White-box module testing can examine the control flow, code and data coverage, and efficiency of the module (13:10–11). The ARTMOS emphasis on modularity and object-orientation is highly conducive to module testing, as described in Section 7.1.1.3.

Once the individual modules have been separately tested, they must be integrated together into a program. Integration testing is a "systematic technique for assembling software while at the same time conducting tests to uncover errors associated with interfacing" (58:299). This approach allows black-box testing of collections of modules to ensure that they work together as specified and that the interfaces are correct. The integration can be either top-down or bottom-up (58:299-301). In top-down integration, modules are

incrementally integrated moving downward through the control hierarchy, starting with the main control module and adding subordinate modules in a breadth-first or depth-first manner. Bottom-up integration begins assembly and testing with atomic (lowest level) modules. As integration moves upward, the need for separate test drivers lessens.

The functional and object-oriented file modules defined in the ARTMOS design serve as natural groupings for the low-level integration of ARTMOS routines. Each file module consists of a set of routines that are intended to work together.

Software testing, also called acceptance testing, tests the entire software system as a unit. This is the final test of the system's compliance with all software requirements (13:18–19). During this phase, black-box testing is performed on the entire system by using representative valid and invalid system inputs to check for correct outputs and error handling. The program is checked for hidden functionality and conformance to the initial design.

7.2 Mapping ARTMOS to Target Architecture

The second objective of this effort was to map the ARTMOS design to a target parallel system and refine the design for the target architecture. In the ARTMOS development, the specification and preliminary design phases proceeded independent of the target architecture. The specific architecture of the target machine, the iPSC/2 hypercube, were applied during the detailed design phase to produce a target-specific mapping of the design.

This section examines the impact of the target system architecture on the ARTMOS detailed design and considers how the hypercube architecture affected that design. Since this objective produced the detailed design of the ARTMOS, this section also traces the final design back to the original requirements specification and looks at some efficiency issues. An analysis of the design methodology as applied to the ARTMOS development is included as well.

7.2.1 Design Impacts of the Target Architecture One major objective during the ARTMOS functional decomposition and the partitioning of functions into modules was to minimize that part of the design that directly interfaces with the hardware or is affected

by the architecture. On the other side of the coin, the remainder of the design should be decoupled ("insulated") from hardware and architecture considerations as much as possible.

This is primarily accomplished in the ARTMOS design through the *Hardware* module, which encapsulates all routines and handlers that directly interface to the local node hardware. Functions are provided for the initialization, resetting, and interrupt handling of local hardware devices, as well as routines for masking and unmasking the processor interrupt levels. By providing all low-level hardware interface functions such as the ones in the *Hardware* module, other portions of the design which use these functions remain unchanged despite changes to the underlying hardware; only the interface routines need modification.

Directly supporting this approach is the separation of the interrupt handling function into hardware handler and OS handler pieces. System interrupts (such as the timer or communications hardware) first initiate hardware handlers in the *Hardware* module, which perform any device-specific actions then call another routine to perform the OS-specific actions. When the receiver hardware interrupts, for example, the RECEIVE INTERRUPT HANDLER is invoked. After performing any hardware-specific resetting of the receiver, the OS handler routine (HANDLE RECEIVE INTERRUPT) is called to process the interrupt. The partitioning of interrupt handling responsibility means that only the hardware-specific handling functions need modified when the hardware changes.

Two routines not included in the *Hardware* module that interface with the processor hardware are SAVE TASK CONTEXT and RESTORE TASK CONTEXT. These functions logically fit into the *TCB* module, which itself requires some modification between targets because of differences in the contexts of different processors.

The most significant impact of the target architecture is on the ARTMOS intertask communications facility, represented by the *Message*, *Message Support*, and *Datagram Queues* modules. Because the implementation of the intertask communications protocols (discussed in Section 6.3.1.3) is completely dependent on the system architecture and communications facility, the communications section *may* need to be redesigned for each

target architecture. Providing the same communications model on, for example, a shared memory machine and a message passing machine cannot reasonably be accomplished with the same communications support routines. Some may need changed, others may be altogether inappropriate. The ARTMOS communications design attempts to minimize the re-design, however.

Other than the hardware interface functions and the communications section, few modifications are required to "port" the design to a new target system. The operations packaged in the *Interrupt* module are hardware independent, with the only needed change being to the INITIALIZE INTERRUPT TABLE routine. This routine sets up the Interrupt Table for the number of hardware interrupts available on the target processor and for the system interrupt vectors needed by the OS. Though hardware-dependent, this routine was left in the *Interrupt* module since it operates on the Interrupt Table object. Likewise, changes may be required in the *Clock* module, since the clock synchronization operation may vary based on the hardware needs.

A final consideration is the potential for "tuning" the operating system for different target architectures or processors. For example, the memory management operations assume that no hardware memory management unit is present. If this device is available, the underlying memory management routines could be altered or removed as appropriate to make more efficient use of the hardware and reduce OS overhead. The intertask communications algorithms could be similarly tailored. In some real-time systems, these optimizations may be critical to meeting system timing requirements. Note that it is the ARTMOS specification and design that support portability; implementations that take advantage of specific hardware configurations are not directly portable to other systems.

7.2.2 The Hypercube and the ARTMOS Design Though the ARTMOS detailed design was targeted towards the iPSC/2 hypercube machine, considerable emphasis was placed on defining the lower-level functions and objects independent of the iPSC/2 specifics. With the exception of the communications section of the design, the emphasis on machine independence was very successful. Very little of the final design, apart from communications, is specific to the iPSC/2 system.

The set of functions packaged in the *Hardware* module, for example, are based on the specific type of hardware devices identified early in the design process as necessary for the ARTMOS regardless of system: timer device(s) and communications hardware. Other generic handlers were specified (Local Hardware Handler and Global Hardware Handler) to perform initialization of any node or system hardware not directly needed by the OS but which must be configured in some way for use. Examples include reading configuration switch data or initializing serial ports. Handlers for the iPSC/2 receiver and transmitter interrupts are provided, but these would be included for completeness regardless: many systems would have similar interrupts and require these handlers, and systems without communications interrupts could simply leave out these handlers.

Since the iPSC/2 is based on the Intel 80386 processor, the TCB structure context area is configured for the 80386's context. The RESTORE TASK CONTEXT and SAVE TASK CONTEXT routines are to be implemented for this context. The TCB remains unchanged otherwise.

The intertask communications facility design for the iPSC/2 is described in detail in Section 6.3.1.3. The design utilizes the iPSC/2 DCM units linking the processor nodes to provide the specified "shared memory" communications model. Datagrams are constructed from task data messages for transmission to the destination node. Operations are therefore needed to queue the datagrams since the DCM hardware may not be able to send the datagram immediately. The datagram approach was used since the hypercube is a loosely-coupled, message passing architecture. In a true shared memory system, the concept of a datagram may be completely inappropriate. The node clock synchronization algorithm selected during the detailed design for the iPSC/2 takes advantage of this communications facility design to acquire distributed clock data.

The types of changes necessary to port this design to other parallel architectures are discussed in Section 7.3.3.

7.2.3 Mapping of Design to Requirements This section documents the traceability of the final ARTMOS design back to the original requirements specification. Appendix E contains a set of tables mapping specific routines in the design onto the specification that

it implements. These tables provide a definition of where specific requirements are met in the design.

For the most part, the design elements flow directly from the original specification. In several cases, however, additional requirements were derived or specification errors were discovered during the design phases that resulted in design elements not found in the specification document. These additions and their rationale are summarized here.

One major change occurs during system initialization. The specification calls for the application system to perform application-specific initialization first, then trigger the ARTMOS for OS initialization and master/slave synchronization of the processors. The design partitions initialization differently. At system start-up, the ARTMOS initializes the local hardware devices and OS data structures, then passes control to the application for application-specific initialization. When the application is ready, it triggers the ARTMOS to perform master/slave processor synchronization and taskset selection. This approach is necessary to ensure that the local processor is in a known state for the application and that the application is fully initialized before multitasking begins. The initialization approach is described in more detail in Section 5.3.3.

The specification defines the requirement for a RETURN FROM INTERRUPT system call, to be used at the end of interrupt service handlers to invoke the scheduler. This allows the scheduler to determine whether the interrupt has caused a higher priority task to become ready. As described in Section 6.3.1.5, however, the requirements for the application to be able to provide its own interrupt handlers and selectively enable and disable interrupts makes the RETURN FROM INTERRUPT call unnecessary. The PROCESS INTERRUPTS routine is invoked for all interrupts, and it calls the appropriate handler if the interrupt is enabled and triggers the scheduler when the handler completes.

For task communications, the specification requires a handler for a receiver interrupt but makes no provision for a transmitter interrupt. The design corrects this oversight, since the iPSC/2 has separate interrupts from the receiver and transmitter hardware. As Section 7.3.3 notes, not all systems have these interrupts, but handlers should be defined for those systems that require them.

The final modification applies to semaphore management. There is no specified requirement to provide a mechanism for the application to bind or notify the OS about its semaphores. The specified semaphore operations would simply act on a semaphore pointed to by the application. Since the OS would have no list of application semaphores, it would have no way of waking tasks in the semaphore queues whose timeout has expired. The BIND SEMAPHORE system call provides this capability.

7.2.4 Performance This section looks at several ARTMOS performance aspects. Though no actual performance data (such as context switch time or interrupt latency) is possible until the ARTMOS is implemented and tested, several comments are appropriate about processor utilization and operating system overhead.

One goal of parallel processing for non-real-time applications is to keep all processors as busy as possible to assure maximum aggregate throughput (92:100). Load balancing is often used to migrate tasks between processors to offload work from busy processors and give idle processors something to do (77:478). Real-time applications introduce other considerations. The designer of a real-time system may wish to add an extra margin of safety by dedicating CPU and other hardware resources to servicing critical resources, even if it means the processor is often idle. The designer may also want to leave processors under-utilized so the workload can be distributed in case of damage or failure to part of the system. "That is a choice real-time designers can and must make based on the nature of their application's functions and how much they are willing to pay for an extra margin of safety" (92:100). Any load balancing schemes applied to real-time systems would need to smart enough to recognize application constraints on processor utilization.

Run-time adjustments to the processor load balance is not permitted in the ARTMOS design. As specified in Section 3.4.1, the application workload is statically partitioned for the appropriate number of processors during application development. The application designer maps out which tasks are assigned to which processor based on specific application requirements and hardware resource availability. This assignment does not change until the designer explicitly modifies it. Processor utilization and load balancing requirements are thus left to the designer.

The overhead imposed by the operating system is difficult to evaluate without actual performance testing. Because the ARTMOS is a prototype design, some algorithms are defined for ease of implementation and clarity rather than maximum efficiency. Every attempt is made to select "good" algorithms for the design, but definition of "optimal" algorithms is beyond the scope of this investigation, particularly when the criteria for optimality vary between applications.

For example, because of the requirement that tasks can specify a limit to the time they wait for semaphores, the OS must scan each semaphore queue whenever the clock is incremented for tasks whose timeout has expired. If the number of semaphores and the length of the queues are small, this method should be adequate, but a better algorithm is needed.

Because of the modular ARTMOS design, however, the designer has the flexibility to configure or "tune" the operating system for specific hardware capabilities and application requirements. This optimization is functionally transparent to the application software so long as the programming model is maintained. The functional operation of the OS does not change, but the designer is able to tune its performance for the target architecture as needed for a specific application.

Though the algorithms implementing the ARTMOS programming model can be modified as appropriate, some additional overhead may be unavoidable on some systems or for some applications. In exchange for this overhead, the designer gains increased programmability, verifiability, maintainability, and portability of applications software.

7.2.5 Design Methodology This section evaluates the ARTMOS design methodology by considering several issues raised during the design process and examining some of the underlying assumptions made by the methodology.

In the ARTMOS development, appropriate methodologies are selected at the start of both the requirements specification and design phases. SADT is selected for specifying the ARTMOS requirements, while in the design phase a hybrid of the DARTS and LVM/OOD real-time software design methodologies is defined. In retrospect, these decisions should not be made separately. One integrated specification and design methodology that meets

all project development requirements should be selected up front. The reason for this assertion is that all of the considered methodologies span both phases anyway. DARTS and LVM/OOD perform requirements specification using data flow diagrams and structured analysis as the first design step. The Ward-Mellor and Hatley-Pirbhai methodologies for real-time software begin with specification and proceed through the software design. Even the non-real-time structured design approach uses the specified data flow or SADT diagrams for the design.

One adaption made to the DARTS and LVM/OOD methods for the ARTMOS design was the use of SADT diagrams rather than data flow diagrams. The reason for this change was that DFDs lack the control flow representation provided by SADT. A real-time operating system such as ARTMOS differs from most real-time systems in that most of its activity is control flow rather than data flow, so it is appropriate to use a specification notation (SADT) that can represent this activity. (25) noted that SADT should work as well as DFD for DARTS, though the control flow information is not needed so early in the design. In the ARTMOS design, however, the early specification of the program control flow simplified the task of defining the process interfaces.

The design methodology evolved from the DARTS and LVM/OOD methods in Section 5.2.1 worked well for the ARTMOS design. The methodology provided a means of defining the generic hardware interface requirements of the ARTMOS, as well as decomposing the ARTMOS into independent processes, analysing the interfaces between them. Though the methodology served its purpose, a significant amount of time was spent fitting the methodology to the project. Both original methods decompose the system into independent, functional processes that pass data and control information through various communications schemes, such as queues, shared variables, and buffers. In the case of LVM/OOD, the processes are mapped to Ada task constructs. A real-time operating system as such the ARTMOS, on the other hand, provides management and communications facilities for tasks such as those developed using LVM/OOD or DARTS. The ARTMOS does not actually consist of multiple tasks; rather, it makes multi-tasking of application tasks possible. The methodologies *worked* for the ARTMOS design, but others may have worked better. Given a real-time operating system such as the ARTMOS, however, the

DARTS or LVM/OOD methodologies could easily and effectively be applied to the development of real-time application software.

7.3 Viability of Approach

The final objective of this research was to evaluate the viability of the design for real-time parallel systems. This section examines the final ARTMOS design with respect to real-time parallel software development, the implementation of the communications model on the target machine, and its viability for real-time parallel architectures other than the target.

7.3.1 Viability for Parallel Programming The first objective of this research was to specify and design a real-time multiprocessor operating system providing the AMCAD programming model. This programming model "presents an abstract, virtual machine to the programmer, allowing the development of applications software independent of the particular target architecture." In this section, the accuracy of this assertion is evaluated.

For a given application, the designer could utilize the ARTMOS programming model to design a parallel solution to the problem that could be executed on a variety of target machines. Since the ARTMOS itself can be "tuned" for the target system, the designer is ensured that the application design executes as efficiently as possible on that machine. In this respect, the above assertion is true. However, the efficiency of the design itself must be considered with respect to the target machine and the scheduling paradigm used.

Vestal (88) examines several scheduling algorithms that may be used for control applications: cyclic rate scheduling, rate monotonic scheduling, and preemptive deadline scheduling. He notes that when using cyclic scheduling, for example, it is often necessary to split tasks with larger periods into multiple segments and inserting the segments into multiple cycles (88:6). Rate monotonic scheduling theoretically allows an arbitrary mix of task periods, but it may be necessary to restrict periods to multiples of some base period to limit the nondeterminism of the scheduler (88:22). Preemptive deadline scheduling may improve processor utilization, but has similar determinism problems because of the overhead and variability of the scheduler. In all three cases mentioned here, the designer

may need to account for scheduler issues in partitioning the application. Tasks may need to be decomposed into subtasks or combined into a larger task to fit into a cyclic schedule or provide the appropriate periods for a rate monotonic scheduler. "It is nearly impossible to write application programs for real-time systems without some knowledge of the attendant executive and the scheduling algorithms in place" (87:61).

The underlying architecture may also impact the efficiency of the design because of the "cost" of task communications. If the target is a shared memory machine with few processors, communications is very fast. For a shared memory machine with many processors or a message passing system, communications overhead may be higher. The impact of the communications overhead is on the granularity of the application tasks. If the overhead of intertask communications is high, the designer may wish to partition the application into larger, more complex tasks with fewer communications requirements. For a system with "cheap" communications, the problem might be decomposed to a lower level to fully exploit its parallelism. Shared memory is not always better than message passing, either. "Multiple processors that contend inefficiently for shared memory over a backplane may behave slower and less predictably than processors that communicate over a high-speed fiberoptic network" (92:98).

The conclusion is that, for real-time systems, the programming model *does* allow software development independent of the specific architecture and scheduling algorithm, but it *cannot* guarantee the efficiency of the design for the target. However, the programming model *does* supports the orderly and consistent development of real-time parallel software by providing coherent, logical task management and communications abstractions regardless of the hardware communications mechanism. The impact of the communications architecture and scheduling paradigm is in the partitioning of the application into cooperating tasks, not in the design and implementation of those tasks.

Research is needed to determine how these scheduling and communications overhead issues can be incorporated into an appropriate real-time parallel software design methodology. One method might be to modify the process selection heuristics to reflect these system issues.

7.3.2 Viability of the Shared Memory Model It is noted above that the section of the ARTMOS design most impacted by the specific architecture of the target machine is that implementing the intertask communications protocol. This is not surprising, as the characteristic that most distinguishes MIMD machines from one another is the way the processor nodes are interconnected. Some consideration must therefore be given to the efficiency of the approach used to implement the specified communications model on the target iPSC/2 communications hardware.

To a certain extent, the efficiency of the communications algorithms cannot be evaluated until the ARTMOS has been implemented, when performance measures are available for comparison. However, several useful conclusions can be reached by comparing aspects of the ARTMOS communications design to a pure message passing approach, such as typically used in distributed architectures like the hypercube.

The ARTMOS implements a shared memory communications model. A task producing data "writes" it to the "shared memory," while a task desiring to consume data "reads" it from the "shared memory." System calls are provided to perform the read and write operations for the tasks. A producer/consumer algorithm protects the integrity of the data exchange.

As explained in detail in Section 6.3.1.3, this communications model is designed using the message passing facility of the iPSC/2 to implement a form of "distributed shared memory." Each processor defines a global data area, which allocates a data structure for each piece of data passed between tasks. A task desiring to write data to the shared memory issues a SEND MESSAGE call (or a SEND MESSAGE AND WAIT call, if the task wishes to wait until the previous data has been consumed). The OS sends the data to the processor node on which the data's consumer task is executing, where the data is written into that processor's global data area. When the consumer task wants the data, it issues a RECEIVE MESSAGE command, which reads the data directly from the global data area. The receive call sends a flag back to the node executing the producer task. With this paradigm, both tasks retain a consistent view of the data's "freshness." because the global data areas on each processor are the same.

The overhead of "writing to shared memory" for this target machine, assuming the consumer task is on another node, thus consists of building a message, waiting until the transmitter hardware is ready, and transmitting the message. When a response is received from the consumer task's node that the data was consumed, a flag is set in the global data area. On the consumer node, the received data message is loaded directly into the the proper data structure in that node's global data area, with a flag set to indicated that the data has been produced. This paradigm is actually *more* efficient than pure message passing approaches, which typically queue up the messages received at the destination node until the receiving task requests the message.

If the consumer task is assigned to the same processor as the producer task, communications overhead is further reduced. The communications routines can "send' the data to the local data area, without having to build or send a datagram. This optimization is transparent to the application tasks involved.

The mapping of the specified shared memory communications model to the iPSC/2 message passing hardware is thus highly successful. The communications overhead is at worst that of message passing, and is potentially more efficient, both in speed and memory usage.

7.3.3 Viability for Different Target Architectures Section 7.2.1 discusses the elements of the ARTMOS design that are system architecture or processor dependent, followed by a description in Section 7.2 2 of how these elements are designed for the iPSC/2 machine. This section considers how the ARTMOS design could be modified for a different type of architecture and the extent of the changes required.

The target system used for the ARTMOS design was the Intel iPSC/2 hypercube, which is a distributed message passing machine. For contrast, this discussion focuses on the other side of the architectural spectrum by looking at a shared memory architecture. Consider a VMEbus-based system consisting of several Motorola 68000 processor cards and a shared memory card, as described in (83) for an unmanned research vehicle avionics suite. In this system, the vehicle control, data link, and surface control reconfiguration software workload is partitioned onto the multiple processors for execution. All task communications

occurs through the shared memory. How would this new architecture affect the ARTMOS design?

First of all, the *Hardware* module specifics are derived primarily from the particular microprocessor and processor card selected. The device initialization and interrupt handling routines are tailored for the specific hardware devices on the processor card, while the processor context is defined for the 68000 microprocessor used on the cards. Likewise, the routines to mask and enable the microprocessor interrupt levels are modified for the 68000's hardware interrupt mechanism.

As noted in Section 7.2.1, the task communications section of the design is the most impacted by the new architecture. Unlike the original ARTMOS design, which maps the shared memory communications model onto a message passing architecture (the hypercube), this version requires that the model be mapped to a physical shared memory card. The shared memory architecture greatly simplifies the communications handling routines because the overhead of message passing (building datagrams from task data messages, queuing the datagrams, transmitting the datagrams, and processing received messages) is avoided. The only OS overhead with the shared memory implementation is that of managing the producer/consumer algorithm and copying the task's data to the shared memory. As before, the RECEIVE MESSAGE command polls on the availability of a consumer task's data. Depending on the number of processors in the system, however, the decrease in communications overhead may be offset by contention for the shared memory.

Because the shared memory communications implementation is so straightforward, most of the communications routines defined in the original design are unnecessary. The datagram queue operations, the build datagram routine, and the receiver and transmitter interrupt handlers can be deleted. The only operations still required are the three message system calls (SEND MESSAGE, SEND MESSAGE AND WAIT, and RECEIVE MESSAGE), the copy buffer function, and a routine to send the consumed flag.

Since the shared memory approach does not pass messages between processors, a different method of synchronizing the individual processor clocks must be devised. Many ways of solving this problem can be implemented using the shared memory.

This section examines how the ARTMOS design originally mapped to a message passing architecture can be mapped to different architecture, based on a physical shared memory. As described above, the modifications for the new target are fairly straightforward, aided by the simpler communications implementation. Further, the shared memory implementation is extremely efficient because of the minimal OS overhead required.

7.4 Summary

This chapter analyzes the specification and design of the ARTMOS with respect to the original objectives. It examines the software engineering aspects of the design, as well as the static and dynamic testing of the design and implementation. The impact of the target hypercube architecture on the design is considered. Finally, the design is analyzed to determine its viability for the architecture-independent development of real-time parallel software and for other parallel architectures. In the next chapter, conclusions are presented and recommendations made for further research.

VIII. Conclusions and Recommendations

This thesis investigation has centered on the specification and design of an operating system for real-time parallel applications which defines an architecture-independent programming model. The ARTMOS development targets complex supervisory control applications, where the entire system is dedicated to performing a particular mission consisting of many lower-level functions. These larger real-time systems are typically found in aerospace vehicle control and avionics, simulation systems, and plant automation processes.

The ARTMOS design provides an architecture-independent programming model. This model can ease the development and test of real-time parallel software by reducing the hardware-specific information required by the programmer, reducing the need to retest the program when ported to a different target, and providing a standard development methodology regardless of the underlying archtiecture. For this methodology to be effective, however, some method of considering scheduling characteristics and communications impact on process granularity must be developed in order to reduce inefficiencies stemming from communications and scheduling contraints.

Further, the intertask communications model is shown to be at least as efficient in terms of memory utilization and processor overhead as communications based strictly on a message passing model, and is in fact as efficient as the underlying architecture allows. For shared memory systems the communications is straightforward and very efficient, constrained only by contention for the shared memory. Likewise, implementation of the model on a message passing system caused no more overhead than typically associated with message passing approaches, perhaps even less since no queuing of datagrams is required at the receiving node.

Much has been accomplished in the ARTMOS development. The core functionality of a real-time multiprocessor operating system is fully specified and designed. The design implements a standard, abstract application interface initially defined in the AMCAD project that enables the development of real-time parallel software independent of the target hardware architecture. The design provides a clearly partitioning between operating system-specific operations and hardware-dependent functions in order to reduce ARTMOS

dependency on any given archtiecture. Finally, the ARTMOS design is mapped to a specific target system, the Intel iPSC/2 hypercube machine.

Much work still remains, however. This section contains suggestions for future research based on the work presented in this thesis. Many such research areas became visible during the ARTMOS development, while others are implementation issues or extensions that are beyond the scope of this thesis. Most of the recommendations below introduce possible extensions concerning the design.

8.1 Implementation

The ARTMOS has proceeded from the initial requirements definition through the specification and design phases in this thesis investigation. The logical next step is to actually implement and test the ARTMOS design on the iPSC/2 hypercube. Many of the benefits of the ARTMOS, including its utility for evaluating OS concepts and considering Ada implementation issues, will not be realized unless the design is actually implemented.

8.2 Alternative Target Implementation

Further benefits can be obtained if the ARTMOS designed is implemented for a second, different target architecture. For comparison purposes, the second target system should utilize shared memory. Two implementations of the design provide the capability to study the efficiency of the programming model on different underlying architectures, the portability of applications software, the ease or difficulty in "porting" the design to the new machine, and so on.

8.3 Ada Implementation

Since the ARTMOS is not being implemented as part of this thesis effort, an implementation language is not selected. Though this decision is best made by the implementor of the design, several recommendations can be made.

The Ada programming language, specifically designed for real-time embedded systems and mandated for all DOD mission-critical software, would seem to be an ideal choice

for implementing the ARTMOS. If Ada is used, however, many additional implementation problems must be resolved. These problems stem from the nature of the Ada run-time system (RTS). The RTS implements many of the Ada features and issues such as dynamic memory management, tasking, run-time error and constraint checking. Other languages leave these features to implement either in the application program or in a real-time operating system.

Since the Ada RTS is in essence a real-time operating system, using Ada to *implement* a real-time operating system introduces problems not usually found with other high-order languages, such as C. These include: inefficiency due to overlap between the RTS and the OS; interfacing the OS to the RTS; developing/compiling Ada applications programs for the OS; and potentially replacing part of the RTS with the OS (requiring access to the RTS source code and extensive knowledge of the compiler implementation). Even worse, these issues must be resolved for each target system and Ada compiler, since each compiler vendor provides a different RTS implementation.

For the above reasons, it is recommended that the initial implementation of the ARTMOS design be in a language *other* than Ada, such as C. The implementation and testing of a real-time operating system can be difficult enough without the above problems. It is felt that if Ada is used, more time will be spent solving the Ada problems than in programming the design itself.

However, it is also recommended that a later implementation of the ARTMOS be done using Ada. Once the design itself has been fully programmed and tested, the Ada implementation can be addressed. If the ARTMOS is to be applied in the military environment, as it should be, the Ada issues must eventually be resolved.

8.4 Fault Tolerance

Multiprocessing for aircraft flight and vehicle control is still a new area. There is much work to be done in the areas of fault tolerance, redundancy management, and reconfiguration. In this context, reconfiguration refers to the redistribution of application tasks onto the remaining healthy processors in case of processor failure or damage to the

system. Reconfiguration can support the graceful degradation of system capability when damage occurs. Multiprocessor architectures are one promising means of accomplishing reconfiguration and fault tolerance, but many questions must still be answered.

While these areas were not requirements for the ARTMOS design, it does not prevent implementation of these areas. The ARTMOS design documented here can be used as a foundation for the construction of higher level operating system capabilities supporting the requirements for fault tolerance, reconfiguration, and redundancy management. Additional research should be performed to examine the impact of these concerns on the ARTMOS design.

8.5 Distributed System Extensions

Another possible extension to the ARTMOS design deals with distributed systems. The ARTMOS is targeted for real-time *parallel processing* applications, where multiple processors cooperate to solve a large problem or collection of problems. Consideration should be given to how an ARTMOS-controlled system or subsystem could be linked to a higher-level system which may or may not be real-time.

One example of such a system is real-time flight simulation, where the simulation computer is networked to other systems for human interface, data collection, imaging, software development, and so on (61). An even more complex example is the Data Management System for the European Space Station, which requires a real-time multitasking environment spread over more than 100 machines on the station (92:97).

Alpha (52) and Chorus (92) are examples of large scale distributed real-time operating systems designed to link many real-time and non-real-time subsystems. Research should be performed to determine how the ARTMOS could interface with such higher-level executives or whether the ARTMOS itself is capable of performing such a role.

8.6 Load Balancing

As noted in Section 7.2.4, load balancing for real-time multiprocessor systems is a complicated problem because of the need to be able to dedicate hardware to critical

tasks. The MTOS operating system, for example, classifies tasks as either global or local. Global tasks reside in shared memory and can execute in whichever processor picks them out of the ready queue. Local tasks are bound to specific processors and have higher priorities to ensure immediate processing of local data (92:102). This approach requires a shared memory architecture, however. The Chorus distributed operating system uses threads (tasks), which execute in "actors." Actors are passive, encapsulated areas of virtual memory and sets of resources such as memory space and data structures. Threads can move among actors on the same processor or move among processors transparently to provide low overhead load balancing. Currently, however, there is no way to lock a thread onto a specific processor (92:102). More research is needed on real-time load balancing and how it applies to the ARTMOS design.

8.7 Real-Time Design Issues

Further research needs to be done into methods for incorporating hardware and operating system issues into the development of real-time software for parallel systems. This method must include such issues as scheduling paradigms, tasking overhead, and hardware execution and communications speeds. Study must also be performed to determine how these factors influence the partitioning of the problem into cooperating processes.

Ideally, application software would be developed independent of these issues, but as considered in Section 7.3.1, realtime software designers rarely have this luxury. Few real-time software design methodologies incorporate any of the above issues to help the designer meet the application requirements.

8.8 Tool Support

A criterion when selecting specification and design methodologies was that they be graphically-oriented and that automated tool support for the methods be available. The SADT and DARTS/LVM/OOD-derived methods selected partially met this criterion. Both methods are based on graphical notations, but only the SADT method, used in the ARTMOS requirements specification, has automated tool support available. Automated

tools supporting the selected design methodology would have saved considerable time in the ARTMOS design phase.

In addition to greatly reducing the time to generate and document the design diagrams, the automated tools could provide invaluable support with respect to consistency checks in module definitions, data flow, and so on. Such tools could reduce the overhead of statically testing the requirements and design as well. One disadvantage of using automated tools supporting a design methodology is that such tools limit the designer's flexibility to adapt the method as appropriate for a given project.

8.9 Testbed Configuration

Section 7.1.1.2 introduced the notion of the ARTMOS serving as a flexible testbed facility for research into operating systems and real-time and parallel software. The design's modular, object-oriented structure provides an ideal environment for experimenting with different algorithms and implementation strategies because it limits the impacts of changes and is clearly partitioned along functional and object boundaries.

A necessary component of such a testbed would be a configuration guide or tool to facilitate the modification of the OS. If a library of suitable replacement modules were developed (schedulers, queue handlers, memory management algorithms, etc.), the configuration tool could assist the user in identifying candidate modifications, selecting alternative approaches from the library, ensuring the consistency of the final product, and assessing the impact of the modifications. Another benefit of such a tool could be to aid the operating system designer in optimizing the design for a particular target system or minimizing the size and overhead of the design by leaving out functionality not needed for a particular system or application.

Bibliography

1. Atlas, Alan and Bill Blundon. "Time To Reach For It All," *UNIX Review:* 49–57 (January 1989).
2. Bamberger, Judy et al. *Distributed Ada Real-Time Kernel Facilities Definition.* CMU/SEI-88-TR-16, July 1988.
3. Bamberger, Judy et al. *Distributed Ada Real-Time Kernel User's Manual, Version 1.0.* CMU/SEI-89-UG-1, February 1989.
4. Ben-Ari, M. *Principles of Concurrent Programming.* Prentice-Hall International, Inc., Englewood Cliffs, NJ, 1982.
5. Bergland, Glenn D. and Ronald D. Gordan. *Tutorial: Software Design Strategies.* Los Angeles, CA: IEEE Computer Society Press, 1981.
6. Bic, Lubomir and Alan C. Shaw. *The Logical Design of Operating Systems, Second Edition.* Prentice-Hall, Inc., Englewood Cliffs NJ, 1988.
7. Black, David L. "Scheduling Support for Concurrency and Parallelism in the Mach Operating System," *IEEE Computer.* 35–43. May 1990.
8. Boehm, Barry W. "Software Engineering," *Tutorial: Software Design Strategies, 2nd Edition.* 35–50. Los Angeles CA: IEEE Computer Society Press, 1981.
9. Booch, Grady. *Software Engineering With Ada, 2nd Edition.* Benjamin/Cummings Publishing Company, Inc., Menlo Park CA, 1987.
10. Bond, John. "Parallel-Processing Concepts Finally Come Together in Real Systems," *Computer Design:* 51–74 (June 1, 1987).
11. Brownlow, Paul. "Get a Handle on Object-Oriented Programming," *EDN:* 265–272 (November 8, 1990).
12. Butler, Ricky W. et al. "Application of a Clock Synchronization Validation Methodology to the SIFT Computer System," *Proceedings of the IEEE 1985 National Aerospace and Electronics Conference.* 194–199. New York: IEEE Press, May 1985.
13. Cadre Technologies Inc. *The Project Manager's Software Testing Guide.* 900616 R10M. Beaverton OR, March 1990.
14. Calingaert, Peter. *Operating System Elements: A User Perspective.* Prentice-Hall, Inc., Englewood Cliffs NJ, 1982.
15. Chandy, K. Mani and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley Publishing Company, Reading MA, 1988.
16. Cheng, Sheng-Chang et al. "Scheduling Algorithms for Hard Real-Time Systems — A Brief Survey," *Hard Real-Time Systems Tutorial.* 150–173. Washington D.C.: IEEE Computer Society Press, 1988.
17. Davis, Dwight B. "Parallel Computers Diverge," *High Technology:* 16–22 (February 1987).

18. DeCegama, Angel L. *The Technology of Parallel Processing: Parallel Processing Architectures and VLSI Hardware.* Prentice-Hall, Inc., Englewood Cliffs NJ, 1989.

19. Emrath, Perry. "Program Laborious," *UNIX Review:* 51–60 (April 1989).

20. Enslow, Philip. *Multiprocessing and Parallel Processing.* John Wiley and Sons, New York, 1974.

21. Enslow, Philip. "What is a 'Distributed' Data Processing System?" *Parallel Processing Tutorial.* 137–144. Washington D.C.: IEEE Computer Society Press, 1984.

22. Fortier, Paul J. *Design of Distributed Operating Systems: Concepts and Technology.* Intertext Publications, Inc., New York, 1986.

23. Freeman, Peter and Anthony I. Wasserman. *Tutorial on Software Design Techniques.* Silver Springs, MD: IEEE Computer Society Press, 1983.

24. Fujii, Marilyn S. "Independent Verification of Highly Reliable Programs," *Tutorial on Software Testing & Validation Techniques, Second Edition.* 116–122. Washington D.C.: IEEE Computer Society Press, 1981.

25. Gomaa, Hassan. "A Software Design Method for Real-Time Systems," *Communications of the ACM:* 938–949 (September 1984).

26. Gomaa, Hassan. "Software Development of Real-Time Systems," *Communications of the ACM:* 657–668 (July 1986).

27. Hartrum, Thomas C. Class handout on $IDEF_0$ distributed in EENG 593, Software Engineering. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, September 1986.

28. Hartum, Thomas C. *$IDEF_0$ Requirements Analysis - EENG 593.* Department of Electrical and Computer Engineering, AFIT. October 11, 1989.

29. Hatley, Derek J. and Imtiaz A. Pirbhai. *Strategies for Real-Time System Specification.* New York, NY: Dorset House Publishers, 1987.

30. Haynes, Leonard S. et al. "A Survey of Highly Parallel Computing," *Computer Architecture Tutorial.* 81–95. Los Angeles CA: IEEE Computer Society Press, 1986.

31. Hornstein, Virgil J. "Parallel Processing Attacks Real-Time World," *Mini-Micro Systems:* 65–77 (December 1986).

32. Houpis, Constantine H. and Gary B. Lamont. *Digital Control Systems: Theory, Hardware, Software.* McGraw-Hill Book Company, New York, 1985.

33. Howden, William E. "A Survey of Static Analysis Methods," *Tutorial on Software Testing & Validation Techniques, Second Edition.* 101–115. Washington D.C.: IEEE Computer Society Press, 1981.

34. Huang, Kai and Fayé A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill Book Company, New York, 1984.

35. Hunter & Ready, Inc. *VRTX/68000 User's Guide* 591313002. April 1986.

36. Intel Corporation. *iMRX II.3 Operating System.* Product Catalog 280618-002. Hillsboro OR, February 1988.

37. Intel Corporation. *iPSC/2 Source Code Internal Product Specification (Release 2.4)* 311595-001. Hillsboro OR, January 1989.

38. Intel Corporation. *iPSC/2 User's Guide* 311532-003. Hillsboro OR, March 1989.

39. Kane, Van R. "Real-Time Kernel, Real-Time Applications," *Electronic Systems Design:* 55–61 (August 1988).

40. Karp, Alan H. "Programming for Parallelism," *Computer:* 43–57 (May 1987).

41. Kieckhafer, Roger M. et al. "The MAFT Architecture for Distributed Fault Tolerance," *IEEE Transactions on Computers, Vol. 37, No 4:* 398–405 (April 1988).

42. Killmon, Peg. "Parallel/Fault-Tolerant Computing," *Government Computer News:* 91–97 (June 12, 1989).

43. Larimer, S. et al. *A Continuously Reconfiguring Multi- Microprocessor Flight Control System.* AFWAL-TR-81-3070, AFWAL/FIGLB, May 1981.

44. Lemansky, Walter John. "Parallel Ada Implementation of a Multiple Model Kalman Filter Tracking System: A Software Engineering Approach," MS Thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, March 1989.

45. Liebowitz, Burt H. and John H. Carson. *Multiple Processor Systems for Real-Time Applications.* Prentice-Hall, Inc., New Jersey, 1985.

46. Lo, Virginia M. "Algorithms for Static Task Assignment and Symmetric Contraction in Distributed Computing Systems," *Proceedings of the 1988 International Conference on Parallel Processing.* 239–244. University Park PA: The Pennsylvania State University Press, 1988.

47. *M68000 User's Manual, Seventh Edition.* Motorola Corporation. Prentice Hall, Englewood Cliffs NJ, 1989.

48. Maekawa, Mamoru et al. *Operating Systems: Advanced Concepts.* The Benjamin/Cummings Publishing Company, Inc., Menlo Park CA, 1987.

49. Muntz, R. R. and E. G. Coffman, Jr. "Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems," *Hard Real-Time Systems Tutorial.* 190–204. Washington D.C.: IEEE Computer Society Press, 1988.

50. Myers, Glenford J. *The Art of Software Testing.* New York, John Wiley & Sons, 1979.

51. Nielsen, Kjell W. and Ken Schumate. *Designing Large Real-Time Systems with Ada.* New York, Multiscience Press, Inc, 1988.

52. Northcutt, J. Duane. *The Alpha Operating System: Requirements and Rationale* Archon Project Technical Report #88011, Carnegie-Mellon University, January 1988.

53. Page-Jones, Meilir. The Practical Guide to Structured Systems Design, Second Edition. Yourdon Press, Englewood Cliffs NJ, 1988.

54. Paker, Y. *Multi-microprocessor Systems.* Academic Press, Inc., London, 1983.

55. Peters, Lawrence J. *Software Design: Methods and Techniques.* Yourdon Press, New York NY, 1981.

56. Peterson, James L. and Abraham Silberschatz. *Operating Systems Concepts, Second Edition.* Addison-Wesley Publishing Company, Reading MA, 1985.

57. Pinkert, James R. and Larry L. Wear. *Operating Systems: Concepts, Policies, and Mechanisms.* Prentice Hall, Englewood Cliffs NJ, 1989.

58. Pressman, Roger S. *Software Engineering: A Practitioner's Approach, Second Edition.* McGraw-Hill Book Company, New York, 1988.

59. Quinn, Michael J. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill Book Company, New York, 1987.

60. Rathje, Edward J. "Smooth Sailing on a "RISCy" C," *ESD: THE Electronic System Design Magazine:* 65–69 (May 1989).

61. Unknown. "Real-Time Ada," *Defense Computing:* 33–38 (July–August 1988).

62. Ross, "Structured Analysis (SA): A Language for Communicating Ideas," *Tutorial on Software Design Techniques.* 96–114. Silver Springs, MD: IEEE Computer Society Press, 1983.

63. Rothman, Joseph. "Putting Software Through Its Paces," *High Performance Systems:* 32–38 (November 1989).

64. Rottman, Michael S. and Daniel B. Thompson. "The AMCAD Real-Time Multiprocessor Operating System," *Proceedings of the IEEE 1989 National Aerospace and Electronics Conference.* 1813–1818. New York: IEEE Press, May 1989.

65. Savitsky, Stephen R. *Real-Time Microprocessor Systems.* Van Nostrand Reinhold Company, New York, 1985.

66. Schindler, Max. "Toward Faster and Portable Real-Time Operating Systems," *Electronic Design:* 31–39 (January 21, 1988).

67. Scott, Michael L. et al. "Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System," *Proceedings of the 1988 International Conference on Parallel Processing.* 255–262. University Park PA: The Pennsylvania State University Press, 1988.

68. Silverthorn, Lee. "Rate-Monotonic Scheduling Ensures Tasks Meet Deadlines," *EDN.* 191–200. October 26, 1990.

69. Small, Charles H. "Real-Time Operating Systems," *EDN:* 115–138 (January 7, 1988).

70. Small, Charles H. "Real-Time Kernels Sprout Into Full-Grown Software Environments," *EDN:* 184–196 (July 6, 1989).

71. Software Components Group. *pSOS-68k Real-Time Multi- Tasking Operating System Kernel User's Manual.* December 1986.

72. Software Components Group. *pRISM-68k Real-Time Interprocessor System Manager User's Manual.* February 1987.

73. Spitzer, Cary R. "All-Digital Jets Are Taking Off," *IEEE Spectrum:* 51–56 (September 1986).

74. Stankovic, John A. and Krithi Ramamritham. "Introduction," *Hard Real-Time Systems Tutorial.* 1–11. Washington D.C.: IEEE Computer Society Press, 1988.

75. Stankovic, John A. et al. "Real-Time Systems: The Next Generation," *Hard Real-Time Systems Tutorial.* 14–37. Washington D.C.: IEEE Computer Society Press, 1988.

76. Stankovic, John A. and Krithi Ramamritham. "The Design of the Spring Kernel," *Hard Real-Time Systems Tutorial.* 371–382. Washington D.C.: IEEE Computer Society Press, 1988.

77. Stone, Harold S. *High Performance Computer Architecture.* Addison-Wesley Publishing Company, Reading MA, 1987.

78. Szkody, Ron et al. *Architecture Specification for PAVE PILLAR Avionics.* AFWAL-TR-87-1114, AFWAL/AAAS-1, January 1987.

79. Tagney, Brendan et al. "Multiprocessors and Multiprocessing in a Distributed System," *Proceedings of the Workshop on the Future Trends of Distributed Computing Systems in the 1990s.* 139–146. Hong Kong: IEEE Computer Society Press, 1988.

80. Tanenbaum, Andrew S. *Operating Systems: Design and Implementation.* Prentice-Hall, Inc., Englewood Cliffs NJ, 1987.

81. Taurus Computer Products. *Introducing Harmony* January 1989.

82. Thompson, D. and R. Bortner. "AF Multiprocessor Flight Control Architecture Developments: CRMMFCS and Beyond," *Proceedings of the IEEE 1986 National Aerospace and Electronics Conference.* 376–382. New York: IEEE Press, May 1986.

83. Thompson, D. *A Multiprocessor Avionics System for an Unmanned Research Vehicle.* AFWAL-TR-88-3003, AFWAL/FIGLB, March 1988.

84. Thompson, D. "Multiprocessor Software Development for an Unmanned Research Vehicle," *Proceedings of the IEEE 1988 National Aerospace and Electronics Conference.* 1512-1516. New York: IEEE Press, May 1988.

85. Thompson, Daniel B. and Michael S. Rottman. "The Use of Nondedicated Redundancy in the AMCAD Fault Tolerant Control System," *Proceedings of the IEEE 1989 National Aerospace and Electronics Conference.* 1770–1774. New York: IEEE Press, May 1989.

86. Thompson, D. "Linear Token Passing Based Bus Interface for a Fault Tolerant Multiprocessor Testbed," *Proceedings of the American Control Conference.* 507–510. August 1989.

87. Tomayko, James. "Applications in Time and Space," *Unix Review, Vol 8 No 9:* 58–63 (September 1990).

88. Vestal, Steve. *Three Scheduling Disciplines for Hard Real-Time Control in Ada.* Honeywell Systems and Research Center, Minneapolis MN, August 1989.

89. Walter, C. J. et al. "MAFT: A Multicomputer Architecture for Fault Tolerance in Real-Time Control Systems," *Proceedings of the Real- Time Systems Symposium.* 133–140. San Diego CA: IEEE Computer Society Press, 1985.

90. Ward, P. T. and S. J. Mellor. *Structured Development for Real-Time Systems,* Yourdon Press, NY, 1986.

91. Westermeier, T. F. and H. E. Hansen. "Recent Digital Technology Advancements And Their Impact On Digital Flight Control Design,"

92. Williams, Tom. "Real-Time Operating Systems Struggle With Multiple Tasks," *Computer Design:* 92–108 (October 1, 1990).

93. Williams, Tom. "Consortium Moves Toward a Real-Time BIOS Standard," *Computer Design:* 38 (December 1, 1990).

94. Wilson, Ron. "Real-Time Executives Take on Newest Processors," *Computer Design:* 88–105 (February 1, 1989).

95. Wind River Systems. *The VxWorks Real-Time Kernel.* 1988.

96. Wisotsky, Steven R. "Extending Structured Analysis to Real-Time Systems," *Defense Computing:* 38–41 (September–October 1989).

97. Yourdan, Edward N. *Real Time Systems Design.* Information and Systems Institute, Cambridge MA, 1967.

98. Yourdan, Edward and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* Prentice-Hall, Inc., Englewood Cliffs NJ, 1979.

99. Zave, Pamela. "An Operational Approach to Requirements Specification for Embedded Systems," *Distributed Software Engineering Tutorial.* 36–56. Washington D.C.: IEEE Computer Society Press, 1982.

Vita

Capt Michael S. Rottman was born on 27 June 1963 in Cincinnati, Ohio. He graduated from Westerville South High School in 1981 and attended the University of Notre Dame, graduating with a Bachelor of Science in Computer Engineering in May 1985. Upon graduating, he received a reserve commission in the USAF and was assigned to Wright-Patterson Air Force Base, in the Flight Control Division of the Air Force Wright Laboratory (WL/FIGL). Since that time he has participated in the design, development, and implementation of the Advanced Multiprocessor Control Architecture Development (AMCAD) project. AMCAD is an in-house project to develop and analyze an advanced fault tolerant multiprocessor architecture targeted for flight control and vehicle management system applications. Capt Rottman has been primarily responsible for the development and testing of the AMCAD Real-Time Multiprocessor Operating System. He is currently the program manager for the AMCAD and Vehicle Management Software Technologies in-house programs. In January 1986, he began the Master's program in Computer Engineering at the Air Force Institute of Technology as a part-time student.

Permanent address: 2017 Partridge Meadows Dr N
Hudson, Ohio 44236