NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.
PROTOTYPING REAL-TIME MULTITASKING SYSTEMS

by

Amjad Junaid B.Sc.

a thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of
Master of Applied Science

Ottawa-Carleton Institute for Electrical Engineering
Department of Electrical Engineering
Faculty of Engineering
University of Ottawa

C Amjad Junaid, Ottawa, Canada, 1990
NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.
I hereby declare that I am the sole author of this thesis.
I authorize the University of Ottawa to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Amjad Junaid

I further authorize the University of Ottawa to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Amjad Junaid
Abstract

Real-time multitasking systems represent a class of complex, dedicated computer systems whose designs require careful analysis and involve several tradeoffs. Two of the main phases in the development of such systems are defining the required functional characteristics of the system, and then translating these requirements into a working design. A substantial amount of progress has been made in systematizing the functional specification phase but the more traditional ad-hoc approach is still largely used in the design phase of these systems. The purpose of this thesis is to show the feasibility of applying development tools and imposing a structured methodology to this design phase of the development process. It is assumed that such a systematic approach to the design phase will reduce development time and cost and result in more reliable and efficient systems. To meet these objectives, this thesis describes the main factors that must be considered in the design of multitasking systems. It also introduces a design prototyping tool that has been specifically designed to gather application specific information pertaining to these factors. Finally, a methodology is proposed that uses this tool to assist in formulating a final design.
Acknowledgements

I would like to express my sincere gratitude to my research supervisor, Dr. Moshe Krieger for all of his support, both technical and personal, provided throughout the course of this work. The numerous technical discussions and arguments proved invaluable in the completion of this thesis. All of the guidance and insights were much appreciated.

Thanks are also due to fellow graduate students and friends Raul San Martin, Robert Joannis and Voja Radonjic. Along with their technical help in figuring out the nuances of these things called computers, their varying personalities and outlooks on life helped to make room B-314 of the Colonel By building a much more interesting and enjoyable place. The friday afternoon tennis sessions and mini-Olympics didn't hurt either.

I would also like to thank my parents and my brothers whose support and encouragement has always been generously forthcoming regardless of my choice of endeavour. Although not always spoken on my part, it has always been appreciated.

Finally, I would like to thank my friends for their support and encouragement. In particular, thanks to Bill and Larry for their humour and down to earth outlooks on life, to Janine for always helping to keep things in perspective, and to Virginia for pulling me from the brink of abandonment on more than one occasion.

I would also like to acknowledge the financial support provided by my employer, Bell-Northern Research (BNR) and by the Natural Sciences and Engineering Research Council of Canada (NSERC).
Contents

Abstract

Acknowledgements

1 Introduction

2 Real Time Systems

  2.1 Real-Time System Characteristics ...................................... 5
      2.1.1 Environmental Events and System Responses ....................... 7
      2.1.2 Real-time Software Structure ...................................... 10
      2.1.3 The Design Problem .................................................. 12
      2.1.4 The Requirement for Multiple Processors ............................ 14
  2.2 Real-time Operating System Requirements .............................. 16
      2.2.1 Resource Management ............................................... 18
      2.2.2 Intertask Synchronization and Communication ................. 20
      2.2.3 Meeting Real-time Constraints ................................... 23
  2.3 Current Operating System Strategies ................................ 26
      2.3.1 Intertask Communication .......................................... 27
      2.3.2 Event Handling ..................................................... 29
      2.3.3 Scheduling .......................................................... 30
  2.4 The Development Process ................................................ 33
      2.4.1 Traditional vs. Modern Development .............................. 34
      2.4.2 System Prototyping ................................................ 38
2.4.3 Requirements for Design Phase Tools .......................... 39

3 The Multitasking System Platform ................................. 42
  3.1 The Hardware Platform ........................................ 43
    3.1.1 Number and Types of Processors ...................... 43
    3.1.2 Hardware Support Facilities ......................... 44
    3.1.3 Processor Interaction Schemes ...................... 44
  3.2 Synchronization and Communication ......................... 49
    3.2.1 Hardware Model and IPC Correspondence ............ 49
    3.2.2 Message Based IPC .................................... 52
    3.2.3 Defining Messaging Primitives .................... 59
  3.3 System Partitioning ........................................ 65
    3.3.1 Distribution of Operating System Functions ....... 65
    3.3.2 Application Software Partitioning ................. 68
  3.4 Event Handling and Response Scheduling ................... 72
    3.4.1 Initial Event Handling ................................ 73
    3.4.2 Difficulties in Real-time Scheduling .............. 74
    3.4.3 Static and Dynamic Scheduling .................... 75

4 System Design Aspects ............................................ 78
  4.1 Design Phase Decisions ..................................... 78
    4.1.1 Impact of Application Software Partitioning ....... 81
    4.1.2 Impact of Hardware Platform ....................... 83
    4.1.3 Impact of Intertask Communications ............... 87
    4.1.4 Impact of Scheduling ................................ 91
  4.2 Statistics and Data Interpretation ....................... 95
    4.2.1 Software Partitioning Factors ...................... 96
    4.2.2 Hardware Platform Factors .......................... 98
    4.2.3 IPC Factors .......................................... 101
    4.2.4 Scheduling Factors .................................. 103
5 The Multitasking Design Assist Simulator

5.1 Motivation and Requirements ........................................ 107
  5.1.1 System Limitations ........................................ 110
5.2 The MDA Architecture ............................................. 111
  5.2.1 The Memory Structure ....................................... 112
  5.2.2 Virtual Processor Interlink Structure ...................... 114
  5.2.3 Virtual Machine Execution .................................. 115
  5.2.4 Kernel Structure ............................................. 118
  5.2.5 Application Structure .................................... 120
5.3 The MDA Language ................................................ 123
  5.3.1 Data Types ................................................ 123
  5.3.2 Instruction Set ............................................. 127
  5.3.3 Event Types ................................................ 131
  5.3.4 MDA Files ................................................ 133
  5.3.5 Expandability of Design .................................... 138
5.4 Collection of Statistics .......................................... 139
  5.4.1 User Collected Statistics .................................. 139
  5.4.2 The Processor Data Group ................................ 142
  5.4.3 The Task Data Group ....................................... 144
  5.4.4 The Queue Data Group ..................................... 146
  5.4.5 The System Data Group .................................... 147
  5.4.6 The Event-Response Data Group ............................ 149
5.5 Execution Path Traces ............................................ 150
5.6 MDA Statistical Primitives ...................................... 155
  5.6.1 Trace Data Structures ................................... 155
  5.6.2 Trace Procedures .......................................... 158

6 The Design Method .................................................. 163
6.1 Requirements of a Telephone Switching Node .................. 165
  6.1.1 Collector Node Hardware .................................. 167
  6.1.2 Collector Node Functionality ............................... 169
6.2 Software Partitioning Analysis ........................................... 172
  6.2.1 Collector Node Software Partitioning ............................. 183
6.3 Hardware Platform Analysis ............................................ 190
6.4 Communication Primitives Analysis ................................. 207
  6.4.1 Semaphore Constructs vs. Messaging Constructs ................. 214
  6.4.2 Synchronous vs. Asynchronous Messaging Primitives .......... 216
  6.4.3 Selective Receive vs. General Receive Primitives ............. 219
6.5 Scheduling Analysis .................................................... 221
  6.5.1 Static Scheduling ................................................ 224
  6.5.2 Dynamic Scheduling ............................................. 240
  6.5.3 Local and Global Scheduling .................................... 256

7 Conclusion ........................................................................... 258

bibliography .......................................................................... 267

A Multitasking Design Assist User Guide .................................. 268
  A.1 MDA Architecture ....................................................... 268
     A.1.1 Virtual CPU Execution ........................................... 268
  A.2 MDA Variables and Data Types ....................................... 271
     A.2.1 Data Types ....................................................... 271
     A.2.2 Variable Definition ............................................. 273
  A.3 Data Addressing Modes ................................................ 276
  A.4 The MDA Language Instruction Set .................................. 276
     A.4.1 Assignment Class ................................................ 277
     A.4.2 Arithmetic Class ............................................... 277
     A.4.3 Stack Class ....................................................... 278
     A.4.4 Queue Class ..................................................... 278
     A.4.5 System Class .................................................... 279
     A.4.6 Control Class ................................................... 281
     A.4.7 I/O Class ......................................................... 282
  A.5 MDA Input Files ......................................................... 284
<table>
<thead>
<tr>
<th>B.2</th>
<th>Hardware Platform Analysis</th>
<th>323</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.2.1</td>
<td>Example Node File</td>
<td>323</td>
</tr>
<tr>
<td>B.2.2</td>
<td>Kernel File</td>
<td>324</td>
</tr>
<tr>
<td>B.2.3</td>
<td>Application Tasks</td>
<td>338</td>
</tr>
<tr>
<td>B.3</td>
<td>IPC Selection Methods</td>
<td>351</td>
</tr>
<tr>
<td>B.3.1</td>
<td>Nonblocking Send With General Receive Kernel</td>
<td>351</td>
</tr>
<tr>
<td>B.3.2</td>
<td>Application Software General Receive Kernel</td>
<td>363</td>
</tr>
<tr>
<td>B.3.3</td>
<td>Blocking Send and Receive Kernel</td>
<td>366</td>
</tr>
<tr>
<td>B.3.4</td>
<td>Application Tasks Using Blocking Kernel</td>
<td>379</td>
</tr>
<tr>
<td>B.3.5</td>
<td>Semaphore Based Kernel</td>
<td>392</td>
</tr>
<tr>
<td>B.3.6</td>
<td>Semaphore Based Application Tasks</td>
<td>403</td>
</tr>
<tr>
<td>B.4</td>
<td>Scheduling Analysis Software</td>
<td>420</td>
</tr>
<tr>
<td>B.4.1</td>
<td>Baseline System Software</td>
<td>420</td>
</tr>
<tr>
<td>B.4.2</td>
<td>Timeslicing Kernel</td>
<td>420</td>
</tr>
<tr>
<td>B.4.3</td>
<td>Response Priority Kernel</td>
<td>420</td>
</tr>
<tr>
<td>B.4.4</td>
<td>Response Priority Application Tasks</td>
<td>436</td>
</tr>
<tr>
<td>B.4.5</td>
<td>Shortest Task First Software</td>
<td>438</td>
</tr>
<tr>
<td>B.4.6</td>
<td>Longest Task First Scheduling Software</td>
<td>440</td>
</tr>
<tr>
<td>B.4.7</td>
<td>Earliest Due Date Scheduling Kernel</td>
<td>440</td>
</tr>
<tr>
<td>B.4.8</td>
<td>Earliest Due Date Application Tasks</td>
<td>455</td>
</tr>
<tr>
<td>B.4.9</td>
<td>Least Slack Time Scheduling Kernel</td>
<td>468</td>
</tr>
<tr>
<td>B.4.10</td>
<td>Least Slack Time Application Tasks</td>
<td>485</td>
</tr>
</tbody>
</table>
List of Tables

6.1 System Event - Response Summary .................................................. 172
6.2 Statistical Results of Various Groupings ......................................... 178
6.3 Task Statistics for Initial Allocation ............................................... 192
6.4 Initial Processor Allocation Results ............................................... 193
6.5 Task Statistics for Second Allocation ............................................. 195
6.6 Second Processor Allocation Results .............................................. 196
6.7 Task Statistics for Third Allocation ............................................... 197
6.8 Third Processor Allocation Results ............................................... 198
6.9 Task Statistics for Fourth Allocation ............................................. 199
6.10 Fourth Processor Allocation Results ............................................. 200
6.11 Task Statistics for Fifth Allocation ............................................. 202
6.12 Fifth Processor Allocation Results .............................................. 203
6.13 Task Statistics for Sixth Allocation ............................................. 204
6.14 Sixth Processor Allocation Results ............................................... 205
6.15 Semaphore Based System Results ............................................... 210
6.16 Synchronous System Results ..................................................... 211
6.17 Receive Specific Asynchronous System Results .............................. 212
6.18 General Receive Asynchronous System Results ............................... 213
6.19 Baseline Scheduling Scheme Results ........................................... 223
6.20 Static Task Priority Scheme 1 Results ......................................... 225
6.21 Static Task Priority Scheme 2 Results ......................................... 229
6.22 Static Response Priority Scheme Results ...................................... 233
6.23 Static Task Priorities With Time Slicing ...................................... 236
6.24 Static Response Priorities With Time Slicing .......................... 237
6.25 Shortest Task First System Results ..................................... 242
6.26 Longest Task First System Results ..................................... 245
6.27 Earliest Due Date System Results ....................................... 249
6.28 System Event - Response Execution Times ............................. 252
6.29 Least Slack Time System Results ....................................... 254
# List of Figures

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>DES Subsets</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Response Timing Parameter Chart</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Modern Development Cycle</td>
<td>37</td>
</tr>
<tr>
<td>3.1</td>
<td>A Shared Memory Model</td>
<td>46</td>
</tr>
<tr>
<td>3.2</td>
<td>The Loosely Coupled Model</td>
<td>47</td>
</tr>
<tr>
<td>3.3</td>
<td>Lock Variable Operation</td>
<td>50</td>
</tr>
<tr>
<td>3.4</td>
<td>Synchronous Message Protocol</td>
<td>53</td>
</tr>
<tr>
<td>3.5</td>
<td>Asynchronous Message Protocol</td>
<td>54</td>
</tr>
<tr>
<td>3.6</td>
<td>Charlotte Buffer Management</td>
<td>56</td>
</tr>
<tr>
<td>3.7</td>
<td>Cause/Wait Synchronization</td>
<td>62</td>
</tr>
<tr>
<td>3.8</td>
<td>Effect of Duplication of Kernel Functions</td>
<td>67</td>
</tr>
<tr>
<td>3.9</td>
<td>Client-Server Interactions</td>
<td>70</td>
</tr>
<tr>
<td>3.10</td>
<td>Mailbox Effects on Client-Server Interactions</td>
<td>71</td>
</tr>
<tr>
<td>3.11</td>
<td>Administrator Task Interactions</td>
<td>72</td>
</tr>
<tr>
<td>4.1</td>
<td>Design Phase Alternatives</td>
<td>80</td>
</tr>
<tr>
<td>4.2</td>
<td>Interdependent Task Allocation</td>
<td>84</td>
</tr>
<tr>
<td>4.3</td>
<td>Statistical Factors</td>
<td>97</td>
</tr>
<tr>
<td>5.1</td>
<td>Virtual Processor Memory Map</td>
<td>115</td>
</tr>
<tr>
<td>5.2</td>
<td>Virtual Processor Interlink Scheme</td>
<td>116</td>
</tr>
<tr>
<td>5.3</td>
<td>Execution Engine Pseudo Code</td>
<td>117</td>
</tr>
<tr>
<td>5.4</td>
<td>Kernel Structure</td>
<td>119</td>
</tr>
</tbody>
</table>
A.3 Example User Defined Type ........................................... 273
A.4 Virtual Processor Memory Map ........................................ 275
A.5 MDA Addressing Modes .................................................. 276
A.6 Node File Syntax .......................................................... 285
A.7 Typical State Machine ..................................................... 288
A.8 Application Task Structure .............................................. 289
A.9 Event File Syntax .......................................................... 292
A.10 MDA Input File Structure ............................................... 293
A.11 Input File Loading Sequence .......................................... 294
A.12 File Structure ............................................................ 295
A.13 User Statistics Gathering ............................................... 301
Chapter 1

Introduction

As the benefits of computer controlled systems have become more and more apparent to a wider range of application engineers, the inevitable has happened in that these systems have become pervasive in all areas of modern society. As these systems gain acceptance in a wider range of applications, there is a push to add more functionality, improve reliability, increase performance, and reduce their cost. However, there is a seeming contradiction in these demands. In many cases, system designers have been resigned to the idea that as system complexity or system performance are increased, reliability can be expected to decline and development and maintenance costs can be expected to increase.

In the past, these sentiments have been justified as most systems were designed in a relatively ad-hoc manner. The application engineer or end user of the system would meet with the designer or design team in order to convey the system requirements. Because this information transfer could not always be conveyed in a precise manner, the specification was prone to inconsistencies. An added complication would occasionally arise if a marketing group played the intermediary between the end user and the designers thus adding an extra opportunity for specifications to become changed or garbled. Once the design team had the requirements, they would use standard hardware and software components employed in other similar systems to come up with a working design. If these standard components were not exactly suitable to the required system, they would be modified to better meet the current needs. As more
functionality was required in the system, more modifications were needed to be made to "make all the pieces fit". Often, these modifications would be unstructured and awkward and would result in inefficient usage of processing time and resources. This in turn, meant a reduction in performance and reliability. Further, development and maintenance costs increased as it became more and more difficult to determine the internal workings of these systems.

In realization of these shortcomings, attempts were made to systematize the entire development process. Much of the work in this area has been devoted to formalizing the process of coming up with a functional requirements specification. The most promising approach taken in this area has been in using formal languages to describe these specifications. The use of a formal language allows the specification to be modelled as a prototype system because the formal language is executable. As such, a relatively wide variety of tools and languages have become available to assist the system developer in producing and refining an executable functional specification [Zav82][KHL88].

However by their very nature, functional specifications lack low level details and are often unusable for determining how the system should be implemented. They are used only for checking the behavior of the proposed system before implementation. Once the functional specification is complete, it is necessary to transform it into the implementation components. While advances in functional specification tools have served to greatly improve the specification phase of the development process, the analogous tools to assist in the transformation to the design specification are comparatively nonexistent.

The design level decisions seem to be made largely based on what hardware and operating system are convenient, readily available, or have been used in the past rather than on what would be best suited for the particular application. Because of this practice, many of the benefits gained from the advances in the functional specification are lost in the design specification. What is needed is a design tool and methodology that is capable of taking a functional specification, transferring it to a
design specification (by implementing it at a lower, more detailed level), and providing the designer with immediate feedback on the consequences of the basic design decisions (number of processors, system partitioning, task to processor allocation, scheduling policy, intertask communication, etc.). In this way, the designer can select a system platform that is tailored to the application at hand and thus allows the system to perform at its highest level. This thesis presents such a tool and design methodology for multitasking systems.

The purpose of the thesis is to provide a feasibility study of applying tools and general systemic methods to the design aspects of complex computer controlled systems. This involves identifying the design aspects common to a wide range of such systems, providing a tool to model and evaluate the particulars of each aspect, and finally integrating these models and evaluations into a concrete design specification process. While the thesis proposes a particular modelling and simulation tool that can be used to assist designers in coming up with a proper design specification, it does not advocate that this tool is the best suited for all applications. Similarly it does not suggest that the proposed design methodology can be followed in all cases nor that following this methodology guarantees an optimal design.

To meet the above goals, chapter 2 introduces and provides the background for the class of computer controlled systems referred to as “real time systems”. These real time systems will be the focus of the design tool and methodology presented in this thesis. As such, chapter 2 also serves to introduce the terminology that will be used throughout the thesis. Specifically, following a general description of real time systems, the special requirements of operating systems used to control real time applications are presented along with a discussion on how some “off the shelf” systems address these requirements. Chapter 2 concludes with a discussion on the difficulties faced in designing such systems and presents an overview of the traditional and modern development approaches.

Chapter 3 narrows the focus of the systems to be discussed by introducing the multitasking platform. The multitasking platform refers to the system design approach involving the division of application software into a number of fairly autonomous
units (tasks) that perform the functional operations under the management of a simple operating system kernel. The multitasking platform is divided into four main components: the hardware platform, system partition, intertask communication system, and event-response scheduling. Each of these system subcomponents is discussed in detail.

Chapter 4 follows this description of the multitasking platform with a discussion on how its individual subcomponents should be designed. This discussion is divided into two parts. The first part identifies the options available and the design level decisions that must be made in the definition of each of the subcomponents for any application. The types of decisions discussed have been chosen so that they are generally applicable to a large class of applications. The second part of the chapter identifies the statistical indicators that can be used in making each of these decisions assuming the proposed system can be modelled. No assumptions are made regarding the manner in which the modelling is done in order to gather these statistics.

Chapter 5 presents a specific prototyping tool (the Multitasking Design Assist or MDA tool) that can be used to model a proposed system in order to gather the statistics discussed in chapter 4. These statistics are then used to make the proper design decisions. Chapter 5 also presents many of the high level details of the tool including the structure of its prototyping language, its run-time environment and the manner in which it gathers statistics.

Chapter 6 has two main purposes. The first is to incorporate the ideas of chapters 3, 4, and 5 into a design methodology that is applicable to most types of real time systems. The second purpose is to illustrate the use of the MDA tool in this methodology. These two goals have been combined by using a concrete example (the low level design of a telephone switching node) as a case study in the use of the prototyping tool and in the definition of the methodology.

Finally, this is followed by a brief conclusion in chapter 7, and by two appendices. Appendix A is a user guide for the MDA tool while appendix B contains the source program listings for the prototypes developed as part of the analysis of chapter 6. These listings are in the prototyping language defined by the MDA tool.
Chapter 2

Real Time Systems

This chapter serves to provide a general background to the concepts used in the design of real-time systems. It describes the characteristics of such systems and presents a description of the current trends in the design of operating system kernels for their control. In addition, the terminology used throughout the thesis is defined in the following sections.

The first section describes the general characteristics and requirements of real-time systems. This is followed by a discussion of the individual components of such systems and how they are affected by the timing constraints placed upon them by the real-time environment. The requirements of an operating system kernel for such an environment are then examined along with some current strategies for their implementation. The chapter concludes with a discussion on the merits of using prototyping methods in the design of real-time systems.

2.1 Real-Time System Characteristics

The terms “real-time systems”, “embedded systems”, and “discrete event systems”, have all come to describe a set of related computer controlled systems. Often these terms are used synonymously with one another but in reality, they each encompass overlapping subsets of such systems.
Discrete event systems can be described as being a class of systems that interact with the environment by accepting external events (generated by the environment) and then performing a set of related actions to provide an appropriate response to each event. One type of discrete event system are transaction based systems. An example of such a system is the automatic teller machines (ATM) located in many financial institutions. These transaction type systems respond to requests initiated by the user in the form of card insertions or keypad input. Each event causes an appropriate response such as the ejection of cash, a transfer of funds, or some other financial transaction. These devices differ slightly from embedded systems in that transaction type systems are man-oriented. That is, their main interactions are with human operators as opposed to embedded systems which interact on a continuous basis with physical processes.

Embedded systems are also referred to as reactive systems or behavior intensive systems. They can then be described as a subset of discrete event systems in which the computer is embedded in another system whose operation it monitors and controls in an ongoing fashion. In the general form of an embedded application, there are two distinct entities, the environment and the discrete event system. The environment is generally a complex dynamic system which includes a multiplicity of physical devices that have to be controlled. The discrete event system is the computer based control system that has to coordinate the operation of a number of the physical components by accepting and responding to external events in a prescribed time order. Process control applications provide good examples of embedded systems. Once process control software is started, it must not take arbitrarily long to run. Also, the program must keep pace with the process that it controls. As a result, the process imposes timing constraints on its control program.

Just as embedded systems are a subset of discrete event systems, real-time systems are a subset of embedded systems as demonstrated in figure 2.1. Real-time systems take the concepts of embedded systems one step further by introducing time as a parameter. Although embedded system must also take time into account by sequencing activities in a given order, in real-time systems, time becomes a critical factor. In
the real-time world, tasks must be performed quickly, at the right time, and in the right sequence. Thus, system correctness depends not only on the logical result of an operation but also on the time at which the result is produced. Avionic control systems are examples of real-time systems.

Formally, a real-time computer system may be defined as "one which controls an environment by receiving data, processing them, and taking action or returning results sufficiently quickly to affect the functioning of the environment at that time" [Sto87]. To deal with the different types of time constraints placed on such systems, they have been subdivided into "hard" and "soft" real-time systems. In hard real-time systems no single failure to meet its demands on time can be tolerated whereas in soft real-time systems, not meeting demands on time can sometimes be tolerated at some cost.

The remainder of this thesis deals with real-time systems as defined in this chapter. Whenever the term embedded system is used, unless otherwise noted, it refers only to that subclass of embedded systems that are considered to be real-time.

2.1.1 Environmental Events and System Responses

Real-time systems, embedded systems, and discrete event systems are all characterized by a number of features that make them distinct from other types of computer
systems. Perhaps their most significant attribute is that in these systems, the computer is not the master entity. Rather, it acts as the slave to the environment which dictates the types and times of responses that must be generated by the computer. Thus the computer must react to the occurrence of sensor detected environmental events, and must respond to them at appropriate instances via actuators.

Because of the nature of real-time applications, the correct timing of responses is critical in providing safe operation of the system. For example, in a process control application, if a response closes a valve after the entire contents of a tank of hazardous chemicals is allowed to escape, the valve might as well not have been closed at all. In this case, the response itself (the closing of the valve) would have been entirely correct. However its mistiming would have rendered it completely useless. Additionally, in some applications, it is essential that things happen close to specified times (i.e. with little jitter) because providing a response too early can be as bad as providing it too late [Gen85].

In many cases, response timing requirements are not as clear cut as this example might indicate. Usually there are many different responses that are required from the system. Some of these may be more critical than others and their degree of urgency may vary with time. In some cases responses need to be produced in a prescribed order at fixed time intervals. These periodic responses are characterized by some period $p$ meaning that they must be produced exactly once every $p$ time units. In contrast, sporadic responses may be required at odd unpredictable times. Because most real-time systems are characterized by the requirements for both types of responses, the embedded controller must also be able to handle both types.

Embedded systems can be specified at various levels of abstraction. At the highest level the system is defined as being composed of event responses, each corresponding to a response to the occurrence of an environmental event. As stated above, the requirement for interactions with the environment gives rise to two types of responses: periodic and sporadic. Unlike the periodic responses, a sporadic response can be invoked at any time as a result of an event occurrence. These types of responses
are initiated at unpredictable times thereby introducing an element of nondeterminism into the system. Periodic responses typically arise from sensor data or control loops while the sporadic load component arises from operator actions or asynchronous events. A real-time system model $M$ is then composed of the set of periodic responses $M_p$ and the set of sporadic response $M_s$.

In real-time systems, each response is assigned a deadline relative to the time that it is triggered. The consequences of missing this deadline will vary depending on whether the response has hard or soft time constraints. If a hard deadline is missed, then the tardiness of the response will cause a critical system error. A tardy response to a soft deadline may cause one of two things to occur. First, the missing of the deadline may render the response useless but still allow the system to survive without appreciable loss in functionality. An example of such a scenario could be the missing of a single periodic temperature reading in a process control application. The second thing that may occur in the case of a tardy soft real-time response is that the response will still be acceptable although it may degrade the system to some degree.

From this discussion, it is evident that each response $i$ is characterized by a number of timing parameters. The nature of the parameters varies depending on whether the response is sporadic or periodic. A sporadic response is characterized by the time at which its triggering event occurs $e_i$, the time at which the event is accepted $a_i$, the time at which the response begins $b_i$, the deadline by which it must complete $d_i$, and its computation time $c_i$. As a result, each sporadic response may be represented as a six tuple $R_i = (i, e_i, a_i, b_i, c_i, d_i)$ [LA90].

Similarly, a periodic response is characterized by its begin time $b_i$, its deadline $d_i$, its period $p_i$, and its computation time $c_i$. The time of event occurrence is implicit in the period of the response. For each periodic response $i$, the following relation must hold $c_i \leq d_i - b_i \leq p_i$.

In both the sporadic and periodic cases, time constraints may be placed on any one or more of these parameters. These time constraints may be either hard or soft. The various timing parameters associated with responses is illustrated in figure 2.2.

The controller is designed to reflect the attributes of the environment so that
Figure 2.2: Response Timing Parameter Chart

an inherent correspondence develops between the environment’s events and the controller’s responses. The special characteristics of the controlled entity have a very strong influence on the design of the system controller.

2.1.2 Real-time Software Structure

In embedded applications responses must be provided to a number of environmental events, many of which may be occurring simultaneously. To handle these conditions, embedded software must be structured and specified differently from other computer applications.

Each system response is composed of a number of atomic entities called activities which are defined as contiguous operations that once started, can be executed from beginning to end without having to wait for data (or other information or resources) from other activities. An event response then corresponds to a set of activities which must be executed in a given precedence relation in order to provide the required response to an external event. In addition, the response may have resource requirements which must be met in order to ensure its correct operation.

Event responses, or more simply responses, can be further categorized as being either simple or complex. A simple response may be either a single activity or be composed of a set of activities that have a purely sequential precedence relationship. For example, let us consider a soft drink bottling process in which empty bottles are
moved along a conveyor to become filled and sealed. At the filling station, the event occurrence would be the arrival of an empty bottle and the response would be the filling of the bottle. The activities might be: stop conveyor, open fill valve, close fill valve, start conveyor. Each of these activities must be done in strict sequence and so the response is simple.

Now consider a multi-jointed robot arm whose end effector must be moved through a three dimensional space and positioned above some object when a button is pressed. In this case the event is the detection of a pressed button and the response is the appropriate arm movement. The individual activities may be: move joint #1 \( x \) degrees, move joint #2 \( y \) degrees, and rotate the gripper \( z \) degrees around its \( z \) coordinate. Each of these activities could be done in sequence but the same response could be achieved much faster if all of the activities were done in parallel. The concurrent nature of activity execution defines a complex response. Another variation of the robot arm may require the arm to move only in a restricted trajectory that requires the individual activities to be done in parallel in a specific complex precedence relation. This type of response in which activities must be done concurrently is also categorized as being complex.

When the abstraction of the response-activity model is ported to a real software environment, the concurrences present in complex responses are used to provide the parallel actions needed by these responses. In addition, concurrences may be exploited to achieve faster response times. The vehicle used to provide concurrency in real-time systems is the task. Tasks are a set of related activities defined in an ordered sequence which define a single (sequential) thread of control within the program. Multiple tasks running concurrently are used in the embedded application to implement the natural parallelism within and between event responses. The task concurrency is achieved by either using multiple processors or by interleaving task execution on a single processor, or both. A task can then be defined as being a self contained block of code that performs a single service for the system.

In the case of the simple response, all activities would thus be placed into a single task. However in the much more common complex responses such as the robot arm,
concurrent activities are placed in different tasks so that they have the potential to run in parallel. Hence the definition of the simple response can also be "a response which is composed of a single task" and that of a complex response is "a response that is composed of multiple tasks".

An embedded program then consists of a set of tasks interacting with the environment and communicating among themselves. Intertask communication is required because in most cases, the tasks comprising the various event responses are not independent of one another. They may need to send data or signal amongst themselves, to synchronize with each other, or to request actions from one another. In most cases, the communication and synchronization between two cooperating tasks is related to the precedence constraints between their constituent activities. These constraints are such that the completion of some activities of one task enables some activities of the other task to be ready for execution.

A large amount of research has been done on the various methods of directly partitioning a system into its constituent tasks in order to maximize concurrency, minimize the overhead associated with intertask communication, and minimize the contention for shared resources between competing tasks [Woo86] [DD86] [Gen85] [Gen81] [Boc77]. A somewhat lesser amount has been similarly devoted to first defining the activities and then grouping them into tasks [KHL88] [PS87] [Sha86]. An effective allocation of activities to tasks must try to achieve a number of goals: maximize system concurrency, minimize intertask communication, and meet the resource needs of all activities. This is only one aspect of the design of real-time systems.

2.1.3 The Design Problem

The development of real-time systems is a fairly complex process which is fully described later in section 2.4. In this introductory section, a few of the peculiarities of real-time design are outlined.

The emphasis in embedded applications is on the physical system that the computer must monitor, respond to, and control. Interacting with the real world which
is outside the computer and thus beyond the programmer's control is a fundamental characteristic of these applications. Since the overall behavior of the embedded system is dictated by the unpredictable behavior of the physical world, a mathematical analysis of system behavior is always incomplete and never completely accurate [Gen85]. This situation can be attributed to three main factors. Firstly, the controlled entity is not easily described by ordinary or partial differential equations because the environment changes in time as a function of complex interactions of timed discrete events. Nevertheless, attempts at such an analysis have been made as in Glynn's Formalism for Discrete Event Systems [Gly89].

Secondly, the embedded controller itself consists of a number of hardware and concurrent software components whose interactions are difficult to model. Still, modelling attempts have been made by representing the environment as a black box and then using various forms of queuing theory and flow models such as Petri Nets to predict controller behavior [Sur89, CMQV89]. While these approaches provide reasonable models, they all have certain drawbacks. For example, while queuing theory does provide a good stochastic foundation, its analysis is useful only for average long term evaluation. It can not begin to do a complete detailed step-by-step system analysis. Petri Nets are better at providing such an analysis, but their behavior is deterministic and therefore unsuitable for the random nature of embedded systems. These methods model either the controller or the environment but they do not close the loop to demonstrate the interactions between the two entities.

The third difficulty lies in the fact that the embedded controller has only limited access points to the environment via a finite number of sensors and actuators. As a result, it can only track some aspects of the real world but can not be in complete control and therefore can not even theoretically provide a provably perfect solution that will account for all environmental states and conditions [Gen85].

While it is not possible to do a full analysis of any but the simplest embedded systems, a feel for system behavior can be gotten through the use of simulations and prototypes. However, because these models are never perfect, additional measures must be taken to ensure a high degree of correct system behavior. One of
these measures is the introduction of multiple processors for increased reliability and performance.

2.1.4 The Requirement for Multiple Processors

Despite the difficulties encountered in their modelling and in proving their correctness, real-time systems must still be extremely reliable because of the nature of the applications that they control. For this reason, these systems must incorporate fault tolerance strategies that ensure fail-safe or fail-soft operation. The most common of these methods is the distribution of control across several processing elements. If one of the processors fail, the others are able to fully or partially compensate for the faulty unit. This distribution of control can be configured in one of several ways. In some cases, all processors are active at all times and when a fault occurs in one of the processors, the remaining units are able to take on the load of the failed unit. This is known as a load sharing mode. Alternately, duplicate processors operate in hot standby mode in which the extra processor mimics the actions of the active processor and takes over operation whenever the active processor develops a fault. In a third possible configuration, several processors are dedicated to performing identical operations. If they come up with different results for the operations, the dissenting processor is overruled by the majority result. The common thread among all of these fault tolerant strategies is that the reliability requirements can only be achieved through the redundancy and reconfigurability provided by multiple processors.

In addition, the performance requirements of real-time or embedded systems of any complexity require multiple processor implementations [Gen85]. Because of the asynchronous and parallel nature of event occurrences in the environment, many embedded systems require real concurrency in order to provide responses in parallel as opposed to apparent concurrency in which responses are provided in an interleaved fashion. This real concurrency can only be achieved through the use of multiple processor systems. When there are many different kinds of external events to which responses are required within specified times, it is far easier to dedicate processors to the different classes of events rather than having to analyze response interactions
within a common single processor schedule [Gen85]. Also, the functional requirements of many embedded systems are such that they require the use of different specialized processors rather than the use of a single high-performance system.

The main difficulties that arise in implementations of multiple processor real-time applications are the movement of information in a timely fashion (i.e. moving required data from one processor to another while still being able to meet response time constraints) as well as software synchronization and scheduling across different processors. The methods provided by the operating system to exchange information between processors are highly dependent on the underlying architecture of the multiple processor system. There are many ways in which real-time multiple processors may be interlinked but they can be broadly categorized into two classes.

One of these classes is known as the shared memory model in which two or more processors access common data. This model may be implemented in one of three ways. There may be a single common memory to which all processors have access or each processor may have its own local memory to which the other processors also have access. The difference between the two is in the number of processors that will be contending for the same memory system at any one time. In the third variation of the shared memory model, each processor may also have a private local memory in addition to the shared global memory.

The other class is referred to as the message passing or distributed model. In these systems, individual processors can not directly access each other’s memories. They must interact through some communications protocol. Unlike the shared memory model, both the sending and receiving processor must be aware of any interprocessor data exchange and both must take an active part in it. To simplify the message passing model somewhat, this discussion assumes the communications protocol ensures error-free point to point communication. This interlink class is also sometimes referred to as “loosely coupled” in contrast to the “tightly coupled” shared memory systems.

While there are significant differences between the two models, they both are
essentially a collection of processing components physically and logically interconnected possessing varying degrees of decentralized system-wide resource control and executing application programs in a coordinated manner [Sha84].

Finally, multiple processor systems are also attractive in that they are modular which makes them easier to implement, upgrade, and maintain [FK83]. As a result real-time systems are usually characterized by multimicroprocessor implementations because they afford the following advantages: high system performance, high availability and reliability, graceful degradation, and ease of modular incremental growth.

The way in which processors of a system are linked, the manner in which the application tasks are allocated to the processors, and the way in which activities are grouped into tasks all have ramifications on the requirements of the real-time operating system used to oversee the application. The nature of these application characteristics is used by the operating system in doing task scheduling and resource management. Thus the structure and specification of the application tasks must be considered when designing intertask communication primitives, doing resource management, and defining a scheduling policy.

The main objective of real-time computing is to meet the individual timing requirements of each of the responses and to make the system as reliable as possible. Proper system partitioning helps in meeting this objective but the main burden of coordinating the individual tasks so that these system objectives can be fully realized is borne by the real-time operating system.

2.2 Real-time Operating System Requirements

Modern general purpose operating systems are designed to provide a multitude of features including protection, security, accounting, reliability, interactive responsiveness, and virtual memory [KRL86]. However, for real-time applications, many of these features are extraneous. When one considers that embedded applications are typically compact and required to provide responses in real-time, it is apparent that the operating system must also be a very small and very efficient package. For this
reason, real-time operating systems are often divided into several functional units which can be individually included in a system as required. These systems typically include a small run time support system known as the kernel which is responsible for providing concurrency in the system, for task creation/deletion, scheduling, and intertask communication and synchronization. These systems may also include an I/O subsystem and a communications subsystem (in loosely coupled applications).

In real-time systems, the hardware and the application software are more tightly intertwined than in regular general-purpose systems due in large part to the specialized I/O devices that must be directly manipulated by real-time applications software. These interactions between the hardware and the application occur via the kernel. The kernel is defined as being the lowest part of the operating system that is directly accessible to the application software and that directly interacts with the underlying hardware. It is a compact "central" subset of an operating system - usually compact enough to be coded onto a single ROM chip. Typically, kernels occupy less than 50 Kbytes of memory while complete operating systems can require several megabytes [Elp89]. The kernel includes the essential features of a viable operating system, but does not include the "outer layers" and utilities. The small size and minimum functionality mean that facilities such as virtual memory, security, large address spaces, and general timesharing are not supported. Such support is unnecessary because these features conflict with the general philosophy of embedded systems. These are compact dedicated systems running trustworthy application software that does not require a complete "hand holding" operating system.

While the kernel is designed to be fairly simple, the overall system may be quite complex when the application is considered. Real-time system complexity depends on both the number of tasks and the degree of interaction among tasks. In most applications, these tend to increase together, and therefore the number of tasks determines complexity. Applications with very many tasks force designers to use multiprocessing strategies to limit the complexity of individual system components and to maintain acceptable event response times. As a result, real-time operating systems must have
the ability to support multiple processor applications based either on the shared memory or loosely coupled interlink schemes. Several common aspects must be considered in providing this support.

Firstly, in multiple processor systems, adding computing power can be done by adding more processors only if the OS has been designed to make use of as many processors as are available. Even so there is a danger that the overhead incurred in coordinating tasks across several processors will reduce much of the benefit gained from the added components. Ideally the system should show essentially linear throughput improvements as processors and related hardware are added.

To achieve this ideal, the operating system must coordinate the application tasks to keep the amount of network traffic or bus contention low by retaining as much locally required information as possible on the local node. Attention must also be given to an efficient communication system, methods for distribution of control, task migration between processors, support for fault tolerance, and the handling of I/O.

2.2.1 Resource Management

In addition to merely providing support for multiple processor implementations, the real-time operating system must also be able to efficiently allocate the processing resources. Because environmental events tend to occur asynchronously and often in parallel, it is necessary that the event responses also be provided in a concurrent fashion. To achieve this end, the real-time OS must support interleaved execution of tasks on each processor in the system in addition to the support for multiple processor task execution. Because of the cost of some resources and as a means of increasing utilization and throughput, resources such as the CPUs, memory, I/O devices and drivers, program segments, and data are shared by these concurrent tasks. The dependencies and requirements for shared resources among concurrent tasks in turn imposes the need for resource management from the OS.

To prevent interference with each other, not more than one operation modifying a shared resource (including a data resource) can be performed at one time. Operations on shared resources must be programmed as critical sections (a set of operations on a
common structure which exclude one another in time) [Sta82]. This means that the operating system must allocate resources among the competing tasks in a manner that prevents interference between tasks but still allows event responses to be produced at their appropriate times.

Resource management techniques in existing operating systems are limited to some form of prioritized task scheduling. That is, each task has a specific (possibly fixed) priority. Whenever a task of higher priority than the currently executing task becomes ready to run, the current task may be preempted in favor of the higher priority task. This type of resource management alone is not enough to guarantee a response's critical timing constraints. Timing correctness entails ensuring that the response(s) to each event is produced by the time it is required by the environment. In order to achieve this goal, it is necessary that some sort of time-driven allocation policies [TWW87], in which the urgency of the resource requirements of a particular task vary according to the time at which they are requested, be applied. Further, the emphasis must be placed on meeting the timing constraints of complete responses and not on their constituent tasks or activities as is the current practice. This approach is required in light of the fact that the environment interacts with the embedded system in terms of events and responses and not in terms of tasks or activities. Most existing systems lose sight of the fact that it is the environment that is being catered to by the embedded system and not vice versa.

In addition to resource management, the management of tasks, I/O devices, memory, and time are also important kernel functions. Task management involves providing means for creating, destroying, querying, modifying and migrating tasks. The dynamic creation and destruction of tasks is considered by many to be important in today's embedded applications because changes in the environment may necessitate changes in the number of active tasks. However this functionality imposes a significant overhead in the operating system. In most cases, the time penalties incurred because of dynamic tasking are too severe for it to be implemented in real-time systems. The alternative is to allocate all possible tasks and the start on all the required processors. While this approach requires more memory, it may be acceptable because
of the reduced time overhead. Similarly, the ability to modify and query task parameters such as priority is necessary in the ever changing system. Task migration is a useful feature in multiple processor systems as it allows better load sharing and hence improved response times and is also a must for system reliability.

Time management is an important operating system function required to meet exact timing requirements of embedded systems. This function includes providing real time clock facilities such as delay/resume primitives, special alarms, timeouts, and the ability to keep loosely coupled processors synchronized with each other. The delay and resume primitives provide application tasks with direct control over their timing requirements while alarms and timeouts are useful for fail-safe and fail-soft software operation. Interprocessor synchronization is very important in multiple processor systems as many activities are related in time but are run on different processors. To ensure the correct activity sequencing, all processors must maintain a common time base.

2.2.2 Intertask Synchronization and Communication

In order that the intertask data dependencies which dictate response timing may be realized, an efficient intertask communication (IPC\(^1\)) facility that can transfer data within real-time constraints is required. These IPC facilities are implemented in a variety of ways and are usually provided in the form of primitive functions.

Intertask communication mechanisms serve three main functions: they allow tasks to interact through explicit exchange of information; they allow tasks to synchronize with each other; and they allow tasks to share resources with mutual exclusion requirements. To provide these functions, real-time operating systems support a number of primitives which the application software may invoke directly. A call to such a primitive may cause the application task to be temporarily blocked from continuing execution until the requested operation is completed. In the tasking model, activities

\(^1\)The acronym IPC actually stands for InterProcess Communication. Many publications use the term process interchangeably with task thus giving rise to IPC. Since IPC is the well known term in the literature, it will also be used in this thesis. However in this context, it should be read as InterTask Communication.
within a task correspond to the sequence of instructions executed between potentially blocking synchronization or communication primitives. These primitives may be implemented through a variety of different concurrent programming mechanisms with each focusing on one or more of the three aspects of task interactions.

In designing mechanisms for exchanging data between tasks, it is important to first examine the interlink scheme of the multiple processor system. The facilities required in a shared memory system are quite different from those in a loosely coupled system. In the shared memory model, the information exchange is accomplished by the data being placed into an area of shared memory from which it is accessed by both tasks using simple read and write operations. In order to ensure the correctness of this transaction, the operating system must provide facilities to ensure that the producing task is allowed to complete its memory operation before the consuming task is granted access to the data. In shared memory systems, this mutual exclusion is provided by semaphores, monitors, or critical region primitives [MOO87].

With loosely coupled real-time systems however, the operating system can not play such a passive role in intertask data exchange. The loosely coupled architecture forces communication between tasks on different processors to be done through a communications network in the form of discrete bundles of information known as messages. (These messages can also be used for communication between tasks on the same processor.) In such architectures, the operating system must provide primitives that hide implementation details and manage the underlying storage and communications media. Typical messaging primitives always include some form of send and receive functions. To present a uniform intertask communication interface to the application software, the operating system does not differentiate between interprocessor and intraprocessor messaging although the underlying implementations are different. Another reason for using the messaging model in all types of communications is that it can directly support a number of multitasking abstractions such as the client-server model, agent pools, and the proprietor model [Gen81].

Synchronization is required in a system for arbitrating access to shared resources and for satisfying precedence relations. It is the synchronization mechanisms that
determine the order in which activities are to be executed within the system. The operating system mechanisms used purely for synchronization purposes come in a variety of forms including semaphores and event sequencers [MOO87]. These mechanisms are provided to the application tasks so that they can impose execution order restrictions on each other. These restrictions are necessary because tasks often need to wait for other tasks to perform specific operations before proceeding.

The third intertask communications function, the ability to share resources which have mutual exclusion requirements, is related to synchronization. These types of resources require that only one task be allowed resource access at a time or one task completes its operation on the resource before it is released. To fulfill this stipulation, tasks must be coordinated so that if one of them is using such a resource, all other tasks will be blocked from its use. To meet these requirements, real-time operating systems provide constructs such as critical regions [MOO87] which selectively or unselectively lock out other tasks.

In most instances, the three functions provided by IPC mechanisms are not independent of each other. Rather there is a logical coupling between synchronization and communication. In order to provide a response, the activities making up the response must be executed in a predefined precedence relation. Also, before any activity is executed, certain other conditions may need to be met. In many cases these activation conditions required for activities to proceed come in the form of data items and/or resources required from other system elements.

This coupling of IPC functions must be reflected in the communication primitives provided by the OS. Many of these primitives explicitly include synchronization with data exchange in some form of a blocking protocol in which either the task sending the data, the task receiving the data, or both are blocked until the data transfer is deemed to be completed. The conditions necessary for completion also vary among the different types of communication primitives. The Harmony primitives send, receive, try_receive, and reply are used to implement a blocking protocol in which the completion of a communication transaction requires a two way handshake. A task invoking the send primitive is blocked from executing until the target task receives
the data and explicitly unblocks the sender by means of the reply primitive. A task wishing to get data from another task calls the receive primitive and becomes blocked until data is sent to it. Alternatively, a receiving task may choose not to be blocked if data is not readily available by using the try_receive primitive. The ADA rendezvous also combines synchronization with data exchange in that the task sending the data can not proceed until it is released by the receiving task.

The main problem with existing communication and synchronization primitives is that they do not explicitly address real-time constraints and their use doesn’t give system designers confidence that the system will meet its real-time needs. It is hard to predict how tasks invoked dynamically will interact with each other and where (and for how long) blocking over resources or data will occur. IPC mechanisms are needed that take a more system-wide approach to synchronization and that consider interactions between entire event responses rather than simply their constituent tasks. IPC primitives must work closely with the operating system’s scheduler in order to best manage system resources and task coordination.

2.2.3 Meeting Real-time Constraints

Because embedded systems typically control time critical applications, it is important that the amount of overhead introduced by the operating system be kept to a minimum. The overhead must be contained by avoiding any unnecessary saving and restoring of task states during context switches and interrupt handling and by minimizing the time spent doing scheduling and executing system calls. In real-time kernels running on today’s microprocessors, context switching speeds of around 100 microseconds and an interrupt response time of around 30 microseconds are typical. This contrasts with context switch times from 350 microseconds to hundreds of milliseconds and also slower interrupt response times for many general purpose time-sharing systems such as Unix [Elp89]. The real-time operating system cost should be localized so that performance of simpler functions is not degraded because of implementing more complex functions. This is similar to the RISC philosophy in the design of microprocessor architectures.
The abandonment of complex functions and the requirements for speed tend to preclude most memory management functions in real-time operating systems. In these systems there is no time for accessing mass storage devices or for doing dynamic task creations and deletions. As a result, the software is always resident in main memory and there is no need for virtual memory support. Therefore, in terms of memory management, the only provisions are for the dynamic allocation and deallocation of memory and for memory protection. To protect the integrity of tasks from accidental corruption, the OS may need to support the memory access functions of a memory management unit.

Flexibility is another important OS attribute for a number of important reasons. Firstly, in many real-time systems unique site tailoring is often required when an application is installed at many different locations. Secondly because of their long lifetimes, real-time applications must have ongoing support for years or decades. This may require modifications to existing applications or perhaps additional functionality.

Finally, it should be emphasized that embedded operating systems are very specialized and they are not designed to provide the facilities required for a software development environment. As a result, cross development of real-time applications is the norm in that the development system is very often different from the target system. However, the development system may still require the services of the run-time OS in order to allow small scale simulations, etc. For this reason, the OS should be able to function in both the development and target environments with little modification. As a result, the OS must be able to function in different system configurations by providing a small set of facilities that are flexible enough to support evolving applications. In addition to being flexible, these functions must also be compact and efficient.

It is evident from this discussion that operating systems designed for use in embedded environments must have special characteristics to make them viable. The attributes presented are summarized below in point form.

1. Allowances for timing constraints to be specified and met in terms of complete event responses instead of their constituent tasks or activities.
2. Support for multiprocessor implementations.
3. Support for the concurrent execution of event responses.
4. Provisions for the management of shared resources.
5. Provisions for intertask communication and synchronization (IPC) primitives suitable for the particular application.
6. Reduced OS size and increased efficiency attained by providing only a limited number of oft-used streamlined features (not general purpose).
7. System portability across sites and installations.

This thesis proposes an integrated approach to the design of application software and selection of kernel functions such as scheduling, intertask communication, and resource allocation. Improved design methods should allow individual event responses to be produced at appropriate times in a flexible and predictable manner. In order to improve on system design, it is necessary to examine some of the kernel policies employed by some existing real-time systems.
2.3 Current Operating System Strategies

The required characteristics of an operating system tend to vary depending on the nature of the application. The choice between a simple run-time kernel and a full operating system implementation usually balances speed against flexibility. Because real-time applications are usually very specialized and because reducing response times is paramount, they are most often implemented using kernels.

In many existing systems, attempts to meet the needs of real-time applications have given rise to kernels that are basically optimized versions of general-purpose operating systems. These systems simply include support for concurrent program execution without considering the specific requirements of the application. An example of this practice is the V-system [Ber86] which supports parallel task execution but is overburdened by features that are extraneous for embedded applications. These include fairness in resource allocation, support for graphics interfaces, and support for secondary storage. While this approach does allow the system to be used for a larger range of concurrent applications, it fails to provide an integrated solution to the problems brought on by the concurrent time-constrained and resource constrained nature of embedded application tasks.

When designing a run time support system for any embedded application, the objectives of the kernel must always be kept in focus. Real-time kernels are employed only to provide the minimum set of facilities required to: support concurrent task execution, support intertask communication, monitor and respond to peripheral devices, and schedule task execution. The first of these (support for concurrency) is a basic attribute of all kernels and is implemented in a more or less uniform manner. All operating systems that support concurrency allow multiple tasks by interleaving them on the available processor(s). However the remaining three kernel attributes (IPC, handling of peripherals, and scheduling) are all handled in a number of different ways by a variety of kernels. Each of these functions is now briefly considered in turn.
2.3.1 Intertask Communication

The nature of the facilities provided for intertask communication varies depending on whether the kernel is designed primarily for loosely coupled systems or for shared memory systems. Systems such as VRTX which are designed for uniprocessor or shared memory multiprocessor implementations make extensive use of shared memory constructs such as semaphores in their intertask communications [Rea86]. When a VRTX task needs to exchange data with another task, it uses the VRTX mailbox or queue constructs. Both of these are based on general counting semaphores in which the data to be exchanged is stored in a common memory to which access is arbitrated through the semaphores. The VRTX kernel provides a number of high level IPC primitives that perform the low level semaphore manipulations for the application tasks.

At the other extreme, kernels designed for use only with loosely coupled architectures such as the V-kernel achieve network transparency by using a message based IPC system [Ber86]. The kernel employs synchronous message passing in which the sender of data waits until the data has been received before continuing execution. The messages are of a fixed length of 8 words and can be exchanged between tasks and also between groups of tasks. Also, to emulate some aspects of the shared memory model, V-kernel messages can be used to give the receiving task access to the memory space of the sender task. The primitives CopyTo and CopyFrom are provided to the receiver task to hide the network communication details that may be required for these operations.

The SPRING kernel, like the V-kernel, is designed for a loosely coupled system architecture and uses messaging for its IPC [SR87]. However, unlike the V-kernel, it uses asynchronous messaging in which neither the sender nor the receiver wait for message reception. The sender is allowed to proceed as soon as the message send primitive completes. Similarly the receiver is allowed to proceed even if the receive primitive does not find a pending message. In either case however, a blocking primitive with a timed wait is also provided. In the case of message reception, a maximum waiting time can be specified so that a task will be blocked until a message
is available or until the specified wait time period has elapsed. In the case of message transmission, the maximum waiting time specifies how long the sending task is willing to wait before a message is received. While this asynchronous approach does increase the amount of concurrency in the system, it also imposes an overhead in that messages now need to be queued by the kernel.

The HARMONY operating system is designed to be independent of the underlying hardware architecture and as a result makes no assumptions about the nature of the physical communications medium. Because it it easier to adapt messaging based IPC to shared memory systems rather than to adapt shared memory primitives to loosely coupled systems, the IPC method used by HARMONY is synchronous message passing [Gen83]. The messaging primitives can easily be implemented using either the shared memory model or the loosely coupled model.

A fifth run-time support system, REX, handles intertask communication in a somewhat different manner still. REX is designed for use in shared memory configurations [BS86]. Intertask communication is done through structures called datasets. Tasks (or processes in REX) are able to allocate and deallocate areas of memory that are used to fulfill their individual memory requirements. These areas are the datasets. Datasets may be shared among several different tasks or among several instances of the same task but they may be accessed only with formal parameters through system primitives. Thus it is these system primitives that arbitrate access to shared data. This is similar to any other shared memory IPC scheme except that in REX, a centralized executive has a priori knowledge of the dataset requirements of each task and so arbitration to the shared data can be simplified by staggering the executions of the competing activities (or tasks in REX).

While all of these IPC schemes serve the purpose of exchanging data throughout their target architectures, they do not address the temporal aspects of intertask communication that are dominant issues in real-time systems [BCMM86]. The SPRING kernel attempts to minimize unnecessary IPC delays by supporting a nonblocking mechanism but it still has no provisions to specify response deadlines and there are no guarantees that interprocess messages will be delivered so that real-time needs
may be met. All of the other kernels discussed use blocking mechanisms. These mechanisms make no attempt at coordinating the communications between tasks in order to minimize the time a task stays blocked, etc.

2.3.2 Event Handling

The second important attribute in kernel design is the handling of peripherals. I/O devices are the means by which the control system interacts with its environment. Interacting with peripheral devices involves two related functions: acknowledging the device (servicing its interrupt) and the managing of the device. Interrupt servicing (saving of system state and collection of volatile data) is done by the kernel while device management is usually left to the application.

The V-kernel handles all devices by treating them as readable or writable files. The system provides operations for opening, reading, writing, querying, and closing devices which can be managed as files. Devices that don't fit the file model yet still need kernel support are handled by the addition of new device specific kernel operations.

Embedded systems often need to interface to a variety of different peripheral devices. Because of this, it is important for a kernel to be flexible enough to accommodate different types of I/O devices without the modifications that are required by the V-kernel. In realization of this fact, HARMONY does not perform any device handling on its own but rather provides the Await.Interrupt primitive that allows an application task to suspend itself until the occurrence of the specified interrupt. So that volatile data associated with the interrupt should not be lost by the time the blocked task is once again prepared for execution, the Await.Interrupt primitive allows the pecification of a specific interrupt handler that is to be invoked as soon as the interrupt is raised. However to avoid conflict with the currently executing task, the interrupt is recognized only if it is of higher or the same priority as the current task; otherwise it is ignored [Gen83].

VRTX provides almost identical interrupt handling services as HARMONY with one exception. VRTX provides the UI.Exit primitive the causes a return from the
interrupt service routine. This primitive explicitly checks to see if the interrupt servicing caused any new tasks to be readied for execution. If this is the case and if the new task has higher priority than the current task, then the current task is immediately preempted in favor of the new higher priority task. If the *UI.Exit* primitive is not used, then the interrupted task is allowed to continue until it executes a blocking primitive.

The method employed by REX for device management is also similar to that of HARMONY and VRTX. Interrupts are always serviced by interrupt specific handlers whose main function is to ensure that data is not lost. If an application task is required to work on the interrupt data, then it is scheduled for execution according to its priority. Because activities are nonpreemptable in REX, the current interrupted activity is always allowed to complete as soon as the interrupt data is saved. In all of VRTX, HARMONY, and REX the kernel simply provides a means for application tasks to be notified of the occurrence of device interrupts. The actual device dependent management of the interrupt is left up to the application [Rea86].

Although the SPRING kernel also follows this philosophy of allowing application tasks to directly control I/O devices, it differs somewhat in the manner in which the model is implemented. I/O devices are categorized as fast or slow depending on their timing needs [SR87]. All slow devices are handled by a single processor running the VRTX kernel with one driver task running for each device. In contrast, each fast I/O device is allocated a dedicated processor running a single driver task. This dedication of resources is done to simplify the scheduling of device handling tasks. The scheduling of interrupt handling routines is often difficult in real-time systems because the assumption that the handling of a particular I/O device is more critical than the currently executing task is not necessarily valid.

2.3.3 Scheduling

The third important kernel function is scheduling. Its main purpose is to satisfy the precedence constraints of each activity in the system and also to satisfy the timing constraints of each response. Further, the scheduler itself must be very efficient so
that it does not introduce a large system overhead. These conflicting requirements make scheduling one of the most complex aspects in the design of a kernel.

The VRTOS, V-kernel, and HARMONY systems all employ a priority based preemptive scheduling scheme. In this method, each task is assigned a priority when it is created. This priority is supposed to reflect the relative urgency of the task. When a number of tasks are eligible to run on any particular processor, the scheduler always selects the one with the highest priority. In addition, whenever a new task is made eligible to run, it may preempt the execution of the current task if the newly ready task has a higher priority. This is to ensure that the highest priority task begins to run as soon as it becomes ready. This is a fairly simple scheduling technique but it leaves it up to the application designer to assign meaningful task priorities.

The REX system also has priority based scheduling but it is implemented in a slightly different manner. In REX, each task's attributes are organized into a task table [BS86]. Among these attributes are the task priority, the sequence relations between the task's component activities, and the period of the task (if it is periodic). Scheduling in REX is divided among three modules: the REX scheduler, the resource allocator, and the dispatcher. The REX scheduler determines when a task (whether periodic or sporadic) becomes ready to run. The REX resource allocator maintains a prioritized list of the ready tasks. The REX dispatcher then assigns processors to the ready tasks based on their priorities and their resource requirements. The resource allocator computes the priority of a task based on the task's initial static priority and on the length of time that the task has been eligible to run. Higher priority is given to those tasks which have been waiting for longer periods.

While all of these systems provide many levels of priorities, they are static priority levels which are appropriate only if the entire system can be completely analyzed during all possible environmental states so that meaningful priorities may be assigned that would be valid under all environmental and system load conditions. Unfortunately such an analysis is infeasible even for the simplest of systems unless all tasks are periodic. In these cases, a predefined cyclic scheduling scheme can be used. Systems containing sporadic tasks can also be scheduled in this manner by limiting the
system to contain only periodic tasks. In this scheme, all critical tasks are made periodic and each is given a guaranteed cyclic time slice on the processor. The major problem with this is that if the task is really sporadic, the time slices allocated to that task are wasted when the task is not active. This is the price of predictability.

In terms of timing constraints, real-time kernels are often divided into two groups: those that support the concept of hard real-time and those that do not. VRTX, the V-kernel, HARMONY, and REX fall into the latter category whereas the SPRING kernel does support hard real-time scheduling.

In order to meet deadlines, traditional hard real-time systems tend to discard any new tasks that can not be scheduled so that the deadlines of already scheduled tasks are met. The SPRING kernel is no different in this respect. It employs a pre-run-time scheduler that schedules all known periodic tasks so that their deadlines will be met. Any sporadic tasks introduced into the system are handled by a small run-time scheduler that attempts to “squeeze them in” if possible [FP88]. Any sporadic tasks that can not be scheduled (fit in among the periodic tasks) by the local node schedulers are given to a system-wide scheduler to determine if it can find a processor that has the time to execute them [SR87]. The assumption is that if a task does not meet its deadline, then there is absolutely no value in running it. For this reason, all hard real-time tasks are periodic while the soft real-time tasks are sporadic. The system is configured to ensure that all periodic tasks can be scheduled to meet their deadlines but any sporadic task initiations may disrupt the predefined scheme.

While this strategy tends to maximize the number of responses that do meet their deadlines, it is unacceptable if the abandoned task(s) served some critical function. This approach assumes that the system consists of a closed set of tasks and a closed set of event possibilities. However, in many complex real-time systems it is not possible to predict all of the task instances that may arise during the lifetime of the system.

The other major problem with scheduling in current operating systems is that all time constraints such as priorities and deadlines are associated with individual tasks [SR87, BCMM86, Sap86] or activities [PS87] except in specific multiactivity
systems [KHL88]. The time constraints of the complete event responses are never explicitly addressed. A certain scheduling algorithm may allow individual task deadlines to be met but some of the higher level responses may still not be produced at the proper times. These methods are appropriate only for simple (single task) responses. In all other cases, it is the responses and not the individual tasks that have real world deadlines to meet. For complex (multiple task) responses new approaches are required for designing real-time schedulers that consider the overall response constraints and the importance of the response to the overall operation of the system. These scheduling policies must be able to dynamically vary the priorities of the schedulable entities rather than attempting to precalculate timing requirements for tasks that display dynamic behavior [SR87]. Finally, the scheduling policy must be applied not only in arbitrating CPU access, but also in allocating system resources, in accessing shared data, and in assigning the priorities themselves. The real-time design methodology proposed in this thesis outlines how various indicators can be used to identify the needs for such scheduling strategies. In hopes of improving this design method, it is necessary to first examine the system development process.

2.4 The Development Process

Traditionally, the system development process has been comprised of four phases:

1. Functional Requirements Analysis

2. Functional Specification

3. Design Specification

4. Implementation

The first phase involved the end user and the system designer defining what types of actions the proposed system was to perform. The second phase involved the development of a requirements specification which typically described the expected system behavior without regard to how it would be attained. This specification was usually
in a natural language such as English. Often, this lack of formalism resulted in a gap between the requirements specification and the eventual design specification. The functional specification phase was followed by the design phase which determined the internal structure of the system. It involved functional decomposition into software modules, selection of the run-time environment (including the system software and the hardware platform), and meeting the required performance constraints. Finally, the fourth phase, implementation, would involve turning the design into executable code, integrating all parts of the system, and doing acceptance testing.

2.4.1 Traditional vs. Modern Development

There are several problems with this traditional approach but the main one is that progress reports on the correctness of the system come very late in the development cycle. An alternative approach involves keeping the same four development phases but making each an iterative cycle of its own with validation through simulations and prototyping at each stage. During the specification phase, a requirements specification is formulated which specifies the system in terms of implementation independent structures which are capable of showing the behavior of the specified system. This is possible only if the specification is described by a formal language which is executable by an interpreter.

The specification phase is distinct from the design phase because the specification language structures are independent of any specific hardware or software resource configurations. They do not refer to any specific run-time environments and are chosen only for modifiability and human comprehension. The main difference between the traditional specification phase and this modern specification phase is that the former treats the development process as a monolithic process whereas the latter requires testing and validation at each development phase by use of new design tools.

The functional specification partitions the system into subunits on the basis of functionality alone. These subunits say nothing about the code structure of the system to be developed. Only in the design phase are there concerns about how the subunits are to be realized on a small number of physical processors. The
specification coming out of the functional phase may be realized by any one of a wide range of possible designs.

Once the functional specification is complete, the structures of the specification must be mapped or transformed into the implementation components. This transformation is the main task of the design phase. According to Zave, a major shortcoming of this type of approach is that the transformation from formal specification to actual implementation is a new and undeveloped technology and that it may be difficult to guide such transformations for even moderately complex systems [Zav84]. The main issues in the design phase consist of selecting the proper hardware and software platforms on which to run the application. In terms of hardware, decisions must be made on the number of processors to be used, the interlink scheme to be used between processors, and identifying any specialized hardware that may be required. The software design decisions that must be made are whether to use a multitasking or multiactivity [KHL88] approach, how to partition the software into schedulable entities, what scheduling scheme should be used, how to assign tasks or activities to processors, and what type of communication and synchronization protocol to use.

The implementation phase in the modern approach is the same as that of the traditional approach in that it involves turning the design into executable code, integrating all parts of the system, and doing acceptance testing. However, this final stage is a much smaller part of the overall modern development process. There are two reasons for this. The first is that the degree of testing at this stage is greatly reduced due to the iterative testing done at each of the functional specification and design specification phases. Once the implementation phase is reached, there is reasonable certainty that the overall design is correct. The only aspect that remains to be tested is that the actual coding and hardware implementation of the design was done properly. The second factor that simplifies the implementation phase is the nature of the design tool presented in this thesis. This tool requires the later refined iterations of the design to be specified in a language whose constructs are very similar to those of popular implementation languages such as PASCAL and C. This is in sharp contrast to many functional specification languages that are geared
towards simplifying user comprehension without paying regards to the limitations of actual implementation languages as in [Zav82]. Adding the middle design tool allows the specification to be mechanically translated to the implementation language.

The modern development cycle is summarized as follows:

1. Requirements Analysis

2. Functional Specification

   (a) Functional description.

   (b) Division into functional subunits.

   (c) Use interpreter/simulator/prototype to verify logical correctness. Revise and repeat if necessary.

3. Design Specification

   (a) Make design decisions regarding system primitives, scheduling, task partitioning and processor allocation, number of processors, processor interlinking.

   (b) Simulate design decisions using design prototyping tool and validate for performance and temporal correctness.

   (c) If results unacceptable, revise design decisions.

4. Implementation

   (a) Coding, hardware development, integration, and testing.

As both the traditional and the modern development scheme demonstrate, any approach to system development can be described as a sequence of decisions leading to a target system. Inevitably, somewhere in this process, it is necessary to back up and remake earlier decisions and repeat subsequent phases. In order to do this properly, it is necessary to gain adequate feedback on the workings of the current design. The most efficient way to achieve this is to construct executable models of the system and observe their workings. This prototyping allows rapid and thorough evaluation
Figure 2.3: Modern Development Cycle
of the system as it stands at a particular time. If the prototype fails to behave as expected, the problems are identified and the functional or design (depending on the phase of development) specifications are modified and a new prototype built to reflect the modifications.

2.4.2 System Prototyping

In order to determine the feasibility of various system aspects it is necessary to construct an executable model of the proposed system. Today the word "prototype" is used in many ways in the literature. In the scope of this thesis, a prototype is a first model on which later versions may be patterned. The early prototypes are used to experiment with and learn about a system prior to commitment. The prototype includes only those attributes necessary for meeting the requirements of the critical parts of the system. The final product of a prototyping activity is a working model that can be used for feasibility study and behavioral specification of a design [Luq89]. The prototype serves only as an aid in analysis and design rather than as a productive system.

In the initial conceptual phase of system design, identifying behavior using accurate prototypes is more efficient than implementing new ideas directly in real applications. The prototype is easier to refine and modify than a full blown implementation if the system initially fails to behave as expected. This is because the interactions within the system are described in terms of simple high level behaviors, and the internal details of many components are not important in the initial stages. As the design develops, these component details may be refined to progressively obtain a more accurate model. Further, prototyping provides continuous feedback information on the system as rapidly and efficiently as possible. So by using a number of different prototypes, enough data can be obtained about the system to allow later implementation in an optimal manner.

In the case of this thesis, the system prototype is used as a modelling and simulation tool that tests the effectiveness of proposed software partitioning methods, processor interlink schemes, and kernel functions under varying conditions.
The potential benefits of prototyping depend critically on the ability to modify the prototype's behavior with substantially less effort than required to modify the full implementation production software. To be able to do this, a good system prototyping language is required. The prototype description language is used to construct a model or prototype of the proposed system. In general, the language should provide the designer with a uniform conceptual framework and a high level description of the system. For operating system kernels, the language should provide a set of building blocks suitable for constructing system components with real-time constraints. Further, the language translator must be able to gather operational statistics so that the designer can determine how the proposed system behaves during its run time.

While prototypes offer special opportunities compared with actual implementations, there are limitations to their applicability. It is true that using system models allows all experimental conditions to be completely under the control of the designer allowing him or her to reproduce a previous result precisely. However, a major problem is that modelling techniques abstract away from the implementation but in fact it is the implementation and the environment which provide the true timing constraints. The prototype can only meet its timing constraints with respect to linearly scaled simulated time [Luq89]. Further, the prototype may not include all aspects of the intended system and it may not be able to handle the full workload of the intended system. Nevertheless, the emphasis in this type of modeling is on proof of concepts rather than on accurate performance measurements. The modern development cycle incorporating the use of prototyping is demonstrated by the flowchart in figure 2.3.

In order to validate the methods presented in this thesis and gauge their effectiveness, a system prototyping language has been designed to facilitate the modeling of real-time kernels and their application tasks.

2.4.3 Requirements for Design Phase Tools

The use of a high level prototyping language allows quick turnaround time in building the new prototype thereby making the modern development cycle feasible. Additionally, the use of a prototype allows exhaustive testing under varying circumstances
throughout the entire development cycle. This type of verification may be able to
detect hidden errors in the design that would otherwise have been missed in the stan-
dard final testing procedures. If the specification is done using a formal language,
then the specification itself can be used as a prototype because it is executable. It
is beneficial to be able to interpret or execute functional specifications because often
they must be developed by successive refinements of understanding with each iter-
tion benefitting from the knowledge gained from the previous ones until the correct
behavior is attained. However by its very nature, the functional prototype lacks low
level details and is often unusable for determining the performance or temporal cor-
rectness of the system or for determining how the system should be implemented.
Functional specification prototypes are used only for checking the behavior of the
proposed system before implementation.

While a relatively wide variety of tools and languages are becoming available
to assist the system developer in producing and refining an executable functional
specification [Zav82][KHL88], the analogous systems to assist in the design phase
of the modern development cycle are nonexistent. The design level decisions such as
number of processors, system partitioning and task to processor allocation, scheduling
policy, etc. seem to be made largely based on what hardware and operating system
are convenient or readily available rather than on what would be best suited for
the particular application. Due to this ad-hoc practice, many of the benefits gained
from the advances in the functional specification phase are lost in the design phase.
What is desperately needed is a design tool and methodology that is capable of
taking a functional specification, implementing it at a lower, more detailed level,
and providing the designer with immediate feedback on the consequences of each of
the above mentioned design decisions. In this way, the designer can select a system
platform that is tailored to the application at hand and thus allows the application to
perform at its highest level. This thesis presents such a tool and design methodology
for message based multitasking systems. Analogous work is also being carried out for
multitasking systems.
System aspects such as software decomposition into functional units, synchronization and communication between these units, the number of processing units, and the interconnection schemes between processing units together comprise the \textit{implementation platform} on which the system is to run. Because most of the decisions made regarding the implementation platform are done during the design phase of the development process, it is imperative that the design specification tool fit the implementation platform. Two alternative platforms can be considered: the multiactivity platform and the multitasking platform. The multiactivity approach uses fine grain schedulable software units called activities whereas the multitasking approach is based on the ideas of communicating finite state machines or Hoare's communicating sequential processes. The next chapter considers in detail the multitasking platform as it is the basis of the proposed design tool.
Chapter 3

The Multitasking System Platform

Due to the complex nature and environmental interactions of real-time systems, many factors must be considered during their development. In these systems, it is difficult to separate the operating system from the application and from the underlying hardware because they are so closely intertwined. Traditionally, application software has been developed around independently designed "off the shelf" operating systems. With the increasing demand for system reliability and added functionality, it has become necessary to update this design philosophy. Instead of designing the application around the hardware and system software (or vice versa), all three must be considered together during the design phase of the development process.

This chapter provides a description of the hardware and software system aspects that must be considered when designing a particular real-time application and how these aspects lend themselves to system prototyping. The following sections proceed to describe how factors such as application software partitioning and hardware configuration affect system functions such as synchronization, communication, and scheduling and how these factors must be integrated in the development of the complete system solution.
3.1 The Hardware Platform

In designing a real-time application, there are three main aspects that must be considered when selecting the appropriate hardware. These are: the number and types of processors that are required to provide adequate performance and reliability, the operating system support facilities provided by the chosen processors and most importantly, the type of interlinking scheme used between the processors.

3.1.1 Number and Types of Processors

Building prototypes allows the designer to model different event arrival rates and to determine with reasonable accuracy, the required number of processors for the expected load. In practice, prototyping gives only a good approximation of this value for a particular application because this number depends on several factors including expected variations in the event arrival rate, the nature of the interlinking mechanism, the number and types of I/O devices, the degree of reliability required through slave, standby, or watchdog processors, and the structure of the operating system. The operating system must be considered because it is the main factor in determining the point at which the overhead associated incorporating an extra processor would outweigh the potential benefits.

The operating system overheads associated with increasing the number of processors are due largely to the increased complexity of task communication, synchronization, and scheduling. Interprocessor communication and scheduling can take up a significant amount of processor time which in turn reduces the application processing capacity of the multiple processor systems. As the number of processors (and consequently the system complexity) increases the result is often a mismatch between the communication capacity of the interlinking medium and the processing capacities of the individual processors. That is, the bottleneck is shifted from the shared communication medium to the individual processors [AA82]. One solution to this problem is to relieve the processors of some of these operating system functions by providing hardware support for communication and scheduling.
3.1.2 Hardware Support Facilities

Hardware support for communications and scheduling functions in the form of hardware protocol handling and multiple contexts through register windows generally results in the speed up of low level communication and scheduling functions [AA82][Sha84]. Another alternative is to dedicate specific processors to handling the majority of operating system functions leaving the rest of the system free to do application processing. In the architecture proposed by Ahuja, a separate processor (SSP - Signalling and Scheduling Processor) is used to manage all intra and inter processor communication. In addition, it also handles the scheduling of activities and their assignment to execution processors. The SSP provides for reliable transport of messages over channels that connect the various processing nodes. Because execution of application software has been separated from the functions of scheduling and communication support, the execution processors do not have to support a large local operating system. However, because all scheduling and message handling pass through a central node for a given number of execution processors, the SSP may become a bottleneck at times of high volume [AA82]. The SPRING kernel proposed by Stankovic also uses a similar scheme to offload system functions to dedicated processors [SR87].

The type of hardware support needed for a specific application is an important aspect of the design phase and must be systematically examined when a functional specification is to be translated into a practical design. However, the most important aspect of the hardware platform that must be considered is the way in which the individual processors are to be physically interlinked. This decision has a great impact not only on the other system hardware aspects (max. number of processors, types of OS hardware support, etc.) but also on the performance and structure of the system software.

3.1.3 Processor Interaction Schemes

When considering multiple processor systems that work in parallel (MIMD execution), it is useful to categorize them by their interactions with other processors. This type of
categorization gives rise to two major system models, the loosely coupled model and the shared memory model as discussed in section 2.1.4. These represent two extremes that have various gradations between them. The choice of the model best suited for a particular application is very important and is dictated by several factors including the expected degree of interactions between tasks, the number of processors required, and the type of IPC scheme used. The effects of the various models on each of these factors is best demonstrated through the use of prototypes. A system prototype is an essential design tool in determining which multiple processor model is best suited for a particular application. Before such a tool can be realized, the characteristics of the models must be defined.

3.1.3.1 The Shared Memory Model

Shared memory model systems are also called multiprocessor systems and are characterized by some variation of the assumption that all data should be accessible to all processors. It is not a strict requirement that all memory be directly accessible to all processors although this is the extreme of the model. In most circumstances, each processor has its own private memory and access to some amount of shared memory. This shared memory can come in the form of local memory and common memory. Access to the local memory is faster than to the common memory because there is generally no need for bus arbitration in the local memory. Each processor is free to access its local memory unless there is an access operation in progress by a remote processor. Thus private memory access is the fastest followed by local memory access. Common memory access or access to another processor's local memory is slowest because of contention. The lack of arbitration hardware for private memory makes it cheaper and its access time more predictable. The private/shared memory compromise is used also because coprocessors may be effective only with fast private memories [Gen85]. Additionally the private memory can make use of a cache without providing additional hardware to solve the cache consistency problems that would occur with the shared memory. However the private/shared memory compromise does create problems with cache consistency in the shared memory and with determining
the best type of addressing scheme to use for accessing the various types of memory. A graphical representation of this shared memory model is given in figure 3.1.

This mixed form of the shared memory model also creates some complications when it comes to deciding on what data items and/or program segments should be placed in which memory. Due to the costs incurred in arbitration hardware and in access time, shared memory is an expensive and thereby scarce resource so only items that must really be shared are located there. This situation creates some problems at run time with respect to dynamic memory allocation. In some instances, it may difficult to determine from which memory space should be allocated if it is not known what data will need to be shared. Consequently, the designer must be able to predict what data and/or code must be shared.

In extreme cases where the processors are more or less autonomous and the shared memory is very limited, it may be necessary to swap items from private to shared memory only when they need to be shared by two or more processors. After the sharing is done, that shared memory area is free to be overwritten. This type of data swapping could however cause a serious overhead in the operating system. In this situation, the loosely coupled model is more appropriate. Also, when memory needs to be shared only among specific processors (ex. between nearest neighbors adjacent on a ring), the loosely coupled model is more appropriate [Gen85].

The decision on whether or not to use some variation of the shared memory model depends heavily on available information on the types of data that must be shared.
in the system, how often that data is shared, and among what processors the data is shared. For systems of any reasonable complexity, it becomes quite arduous if not impossible to keep track of all the data interdependencies without some automated design aid such as that presented subsequent chapters.

3.1.3.2 The Loosely Coupled Model

Systems using the loosely coupled model are also referred to as multicomputers. In this model, the individual processors are not able to directly access each other's memory. Instead, they have to communicate through some kind of communications system that requires the active participation of both the sending and the receiving processors. This communication system consists of a high-level data transfer protocol over an interconnection network. The model assumes that the communications system is completely reliable but no assumptions are made about its nature. The interconnection network could be a serial or parallel link, bus couplers connecting processors' local buses, small blocks of shared memory, or contentionless LAN hardware [Gen85]. However, the common assumption is that the communications system ensures error-free communication. A graphical representation of the loosely coupled model is given in figure 3.2. The implications of this setup is that because both processors are active in the communication, performance is slower. In order to offset this degradation, the software must be designed differently than in the shared memory model.

In the loosely coupled model, an active software module must be provided to handle the reception of communications and to determine when interprocessor requests
need to be made. The complexity of this module varies depending on the sophistication of the communications system. The communications module may be required to cache data locally so that redundant remote requests are avoided or instead of requesting individual items, blocks of relevant data may be requested at one time in anticipation of their subsequent use. In most cases, this communication module will be part of the operating system software but its design will be influenced by the nature of the application software. For instance, the size of the block or cache area will be determined largely by the sizes of the data structures used by the application. Overall, this type of planning of communications leads to a more complex program than that required by the shared memory model.

Although the loosely coupled model adds a degree of complexity to the system software structure, it does have its redeeming features. Its main advantage over the shared memory model is that it reduces contention for common memory so that although individual processors may be degraded, overall system performance is improved. This is especially true if data is shared only between specific processors (ex. nearest neighbors). With a monolithic shared memory, all processors may be left waiting for one set of interprocessor interactions to complete. This is not the case in the loosely coupled model as each set of interactions can be carried out concurrently if the interconnection network allows. Additionally, the loosely coupled model makes it easier to incorporate heterogenous processors into the system as not all the processors' memory accesses need to be done in the same way.

The loosely coupled model's assumption of an error free interprocessor communication path may appear to be somewhat artificial in that communication errors not corrected by the hardware remain undetected. However the loosely coupled model typically limits itself to simple discrete message communications. Any logical errors in received messages are detected by the application software which is in the best position to decide on an appropriate course of action.

At this stage, it is necessary to differentiate between the loosely coupled model and computer network systems used in distributed computing. Unlike the loosely coupled model, the individual processors in the network are not considered to be parts of a
single machine working on a single job. They are individual units that can be started or stopped autonomously. Although computer networks have fairly widespread use, their communications overheads and unpredictability make them impractical for use in most real-time applications.

The implications of the system design decisions made in terms of hardware configuration come up time and time again in other areas of the design phase. The hardware platform has a major influence on software partitioning, response scheduling, event handling, and especially on intertask communication and synchronization. These influences are discussed in the following sections.

3.2 Synchronization and Communication

In real-time systems using the multitasking platform, individual tasks are allowed to proceed in parallel. To avoid the problems associated with concurrent access to shared resources and data, and to ensure proper precedence sequencing between the tasks, there is a need for synchronization between them. Also, since the tasks in the system are never completely autonomous, facilities must be provided to allow some kind of intertask communication. In contrast to the multiactivity platform, the multitasking platform has no centralized controller that explicitly performs all of the synchronization and communication duties. Rather, it is left to the application tasks themselves to determine when and with what other task(s) they need to synchronize or exchange data. To facilitate this process, the operating system must provide utilities or primitives that can be used by the tasks. The nature of these primitives is influenced by a number of factors including the chosen hardware platform.

3.2.1 Hardware Model and IPC Correspondence

The characteristics of the proposed IPC scheme are influenced to a large extent by whether the hardware platform is loosely or tightly coupled. It should be noted that the hardware platform influences only interprocessor communications because communications between tasks on the same processor do not require any data movement
\( r \) - An Internal Register
\( L \) - Lock Variable

Loop:
\[
\begin{align*}
\text{TAS} & \ (r, \ L) \quad ; \ r \leftarrow L \text{ and } L \leftarrow 1 \\
\text{CMP} & \ (r, \ #0) \\
\text{JNZ} & \ \text{LOOP} \quad ; \text{Busy wait if locked}
\end{align*}
\]

\{Access Shared Data\}

outside the processor's own memory.

Using shared memory for interprocessor communications allows efficient data exchange between units of the system because the time required to copy data from one processor's private memory to the other's is eliminated. However, in order to avoid damage to the data through interrupted memory accesses, the common memory used for exchanging data must be protected as a shared resource. Using shared memory immediately suggests doing the required synchronization using semaphores because they are, by their nature, shared memory constructs. Semaphore operations in a shared memory environment require a global lock variable which is checked by all processors before manipulating the semaphore data structures. Mutual exclusion on the lock variable is provided by some sort of atomic test and set (TAS) instruction that allows an indivisible read/modify operation. A processor is prevented from manipulating the semaphore until the currently accessing processor is finished. This process is illustrated in figure 3.3.

However, in a real-time environment, even this type of communication requires more than just shared memory between the interacting processors. If a process blocked on one processor is to avoid busy waiting on a semaphore variable, interprocessor interrupts are needed in order for one processor to signal the other. If there are more than two processors sharing the same memory and more than one is waiting on a particular semaphore variable, the signalling processor must determine which
waiting processor is to be interrupted. As a result, system overhead and complexity rise as the number of processors is increased in this scheme. Oftentimes it becomes necessary to route all such synchronization interrupts through some master processor which can be interrupted by any processor and which can also interrupt any other processor. Although this provides a simpler, more uniform interface to the communicating processors, the centralized signalling processor could become a bottleneck during periods of high communications traffic.

Inter-task communication using shared memory techniques is useful in an environment where the task interactions are limited either to tasks on the same processor or to only a few communicating partners for tasks on different processors. However, as individual tasks are required to communicate with many different partners on different processors, the scheme can become quite problematic. While all communication paradigms using shared memory must contend with this issue of synchronizing memory access, the situation can be relieved somewhat by providing distinct areas of shared memory to be used by specific communicating partners thereby reducing the number of competing tasks. This process can be made more systematic by overlaying a message passing paradigm on the shared memory model. While the underlying interaction structure remains the same, all communications occur via discrete data packets (messages) which are managed by the operating system. This presents the communicating partners with a standard interface as all the low level synchronizations are performed by the OS primitives. Although the nonessential messaging layer adds some degree of overhead in shared memory communications, in systems with large numbers of tasks, it pays dividends in terms of application simplicity and uniformity.

A major benefit of message based communication is that the same paradigm can also be used in loosely coupled systems. In fact, messaging is a natural part of such systems. In loosely coupled systems, for interprocessor communications, messages must be copied from the sending processor to the receiving processor. This overhead is missing in the tightly coupled systems but the compensating factor is that there are no memory contention problems in the loosely coupled model (although the bandwidth of the interconnection medium may become a limiting factor). Because the message
passing concept is applicable to both of the interlink models and because it enjoys widespread popularity in existing real-time systems, it will be the main focus of subsequent discussion.

### 3.2.2 Message Based IPC

A message based communication system has some distinct advantages over semaphore based or procedural (monitor) based communications. Because messages are treated as independent pieces of data that can be transferred throughout the system tasks, messaging provides for good modularity and process independence thereby enhancing software understandability and maintenance. Messages allow a great flexibility in the type of sequence control flow that can be implemented as opposed to the standard procedural flow of control. The order in which messages are sent, received, or replied to can be easily modified depending on individual system circumstances.

Although the messaging concept is based upon the standard principle that communication is done via discrete packets of data that are logically or physically transferred between two or more entities in a system, there are many variations of this central theme. When a system is to be designed using message passing IPC, several design phase decisions must be made about what types of message passing characteristics are appropriate for the intended system. Decisions must be made about task synchronization for IPC, the means of identifying communicating partners, and the format that the messages should take.

#### 3.2.2.1 Synchronous and Asynchronous Messaging

Every message communication involves at least two communicating partners: the sender and the receiver. Because tasks are relatively independent entities, the receiver can not know when it is about to receive a message. Similarly, the sender does not know when its message will be acted upon. Depending on the type of system desired, some type of synchronization between the sender and the receiver may be required. This situation has given rise to two basic modes of message based communication: synchronous and asynchronous.
In synchronous communications, the sending task is made to wait either until the receiving task has received the message or until the receiving task explicitly frees the sender via an acknowledgement. This type of protocol (see figure 3.4) assures the sender that its message has actually been accepted when it resumes execution. Additionally, because the sender is blocked until the message has been received, the sender can not continue to send subsequent messages and thereby possibly over-run the receiver. This type of flow control is implicit in synchronous systems. Its effect is that it eliminates the need for buffering of messages. The major disadvantage of this approach is that it limits the degree of concurrency. In its extreme form, the system may behave in a manner similar to a sequential procedural system in that tasks are constantly waiting for their message requests to be completed by other tasks and are prevented from doing any useful work.

Asynchronous communication can be described as message exchange without acknowledgement. The sending task is no longer blocked waiting for the receiving task to accept its message (figure 3.5). This allows the sender to proceed with other operations in parallel with the servicing of its message. Because there is no task synchronization in this approach, the sender has no assurance that its request message has been acted upon. In these cases to provide tolerance for lost messages, the sender may retransmit messages to receivers which may then repeat their execution. This
duplication of actions should be avoided in general but may be occasionally necessary in the asynchronous system. The situation can be tolerated only if the application is designed such that the receiver's multiple executions have the same effect as a single execution. If this is not feasible, the communication mechanism must ensure that only one execution is performed. Also, the lack of flow control in this approach creates the need for buffering of messages that have been sent but not yet accepted. Thus the asynchronous approach provides a high degree of concurrency and also reduces message traffic by eliminating the need for acknowledgement messages. The price to be paid for this added concurrency are polling (in the case of nonblocking receive) and the overhead associated with buffering.

Even in synchronous systems (as well as in asynchronous), messages must be stored somewhere between the time they are sent and received. This buffering can be done either in the operating system or in the users space. Buffering in the operating system rather than assigning individual buffers to individual tasks in user space has the advantage of flexibility in handling uneven buffer usage. For instance if a particular task is involved in many communications, its user buffers will be quickly filled up while another task which is noncommunicating at the time will have idle user buffers. This waste of buffer space can be avoided by kernel buffering.

A major disadvantage of kernel buffers is the allocation problem that arises when
kernel buffers are depleted. In an asynchronous environment, a single overactive task can quickly use up all of the kernel buffers thereby starving other perhaps equally important tasks. Buffer allocation policies must somehow limit the unrestrained production of messages. One way to do this is to limit the number of buffers a single task may use at any one time. This method works best if all of the tasks have similar buffer requirements. The buffer requirements of individual tasks can be predicted from system prototypes. These types of design phase tools can also be used to determine what type of buffering strategy and what type of synchronization strategy is most applicable to a particular application.

When internode message communications are considered, the issue of buffering becomes more involved. In internode communications, the kernel must do some sort of internal buffering of messages to account for the delay during physical reception and subsequent placement in the user buffer. To allow this, the kernel has to allocate a small number of internal buffers to store messages incoming from other nodes.

Because the number of kernel buffers is limited, some type of allocation policy must be employed to arbitrate their use. The Charlotte system [ACF84] presents a good example of such a policy. In Charlotte, when an incoming message is received by the kernel, the internal buffer is emptied and freed right away if a receive call is pending for the incoming message. Otherwise, the full internal buffer is marked as being free but it is placed at the end of the free internal buffer queue and will be overwritten only if there are no other free buffers available. Thus there is a good chance that the buffer information will still be available intact when the matching receive request does occur. This process is illustrated in figure 3.6. If the buffer has been overwritten the originating kernel is notified that the message was not received and the originating kernel must retransmit the message. With this policy, the kernel guarantees successful message reception but incurs some retransmission overhead. An alternative would be to simply notify the receiving task that it has lost a message and allow it to decide whether or not retransmission is warranted. The optimal number of kernel buffers depends largely on the application software and may best be determined by using a prototype of the application. Also, the prototype is essential in determining
3 Kernel Buffers

A  
Incoming MSG put in free buffer

B  
Incoming MSG put in free buffer

C  
Incoming MSG put in free buffer

D  
Incoming MSG overwrites oldest full buffer

E  
MSG accepted by task

F  
Incoming MSG put in free buffer

Figure 3.6: Charlotte Buffer Management

the effectiveness of any buffer allocation policy for a particular application.

3.2.2.2 Identification of Communication Partners

Another design phase decision that must be made when using a message based IPC system involves the identification or addressing of the communicating tasks. There are several options when deciding on an addressing scheme. In one approach the communication relationships between tasks can be strictly restricted to a one to one relationship and is established when the tasks are created. This simplifies addressing in that the source and destination tasks do not have to be identified at run-time as all of the connections are “nailed-up”. Of course this is a very limiting approach in that any particular task is allowed to send only to one other task and receive from one other task.
A somewhat more flexible approach allows the specification of a source and destination task pair to be used until a subsequent selection is made. This approach is useful if the communication partners are liable to change but not at a rapid rate. It incorporates the simplicity benefits of static addressing but is not as restrictive. If the application requires that few messages be exchanged with various different tasks, rather than numerous messages or an uninterrupted sequence of messages with the same correspondent, then for each message the communicating partners need to be specified explicitly (dynamic addressing). Usually, it is not necessary for the receiver to specify acceptable senders. This receive-any paradigm allows any task to use the services of another task. Similarly, a broadcast send can be used that doesn't specify a receiver and the message is received by the first task (perhaps in some set) attempting to receive or by all tasks.

The desired addressing characteristics are thus heavily dependent on the nature of the expected communications. The type of task interactions that can be expected for a particular application are an important design aspect that can be determined from data collected during a system prototyping activity.

If a dynamic addressing approach is used, another aspect of addressing that must be considered is obtaining and using the name of the correspondent task. This becomes especially important in an environment which is required to support dynamic task creation and deletion where the correspondent identifiers may not remain constant over time. In this case, it may be necessary to introduce a librarian task that maps services to current task names.

An alternative to using task names as addresses is to use links or channels instead. These links may be dynamically established between tasks and the send destinations refer to the links rather than the actual task identifiers [Gen81]. The advantages of directing messages to task names rather than links is the simplicity of usage, the minimum kernel support in not having to maintain tables of dynamic virtual communications channels, and the possibility of compile-time consistency checking. The disadvantages are its difficulty supporting replicated or dynamic tasks and lack of protection in that any task can communicate with any other task at any time.
Although this situation may be desirable in some applications, it could for example allow low priority tasks to tie up a service task when it is needed by more urgent users.

To alleviate this situation, some systems may require the ability to have multiple queues for each task. In this case, the sender must specify the queue in addition to the destination task and the receiver must specify the queue from which it wishes to receive a message. This may be useful in cases where one message must take precedence over other waiting messages. However in this situation, the designer must consider whether the same result could be achieved more efficiently if implemented with receive-specific addressing or by using separate tasks to receive different classes of messages; for example, by having higher priority messages handled by the higher priority task. Again in this situation, a design prototyping tool such as the one presented in the next chapters is invaluable in deciding whether multiple tiered messages are warranted, how they should be implemented, and how their priorities should be assigned.

3.2.2.3 Message Format

Choosing the precise syntax and semantics for message communication is difficult because unlike the existing models for message protocol or addressing, there are no standards for message format. The physical message typically consists of three fields: header, msg text body, and the trailer. The header typically consists of routing information, and the trailer is used for error detection and correction of the message body. It is the text body that contains the actual data that it to be exchanged between the communicating partners.

A major consideration with respect to the message body is deciding whether the messages should be of a fixed length or of variable length. There are implementation advantages to using fixed length messages especially when messages must be moved between different address spaces such as those between different processors without shared memory or between different memory maps on a single machine. With fixed size messages, there is no need to extract the length of the message from the header.
as the same number of bytes are always transferred. This provides for a fixed reliable transfer time for all messages. Similarly, if the application uses mostly very short messages, it can be much faster to move a fixed number of words which may not all be needed than to incur the overhead of variable sized messages. Fixed length messages also avoid requiring the receiving task to know how much space is required to store the received message [Gen81]. One of the advantages of variable length messages is that they can reduce the number of messages in the system by being able to accommodate large messages. In the fixed sized format, such messages would need more than one message transfer.

It is apparent that many factors must be considered in the design of a messaging system for a particular application. The next section describes how these factors can be amalgamated into discrete messaging primitives.

3.2.3 Defining Messaging Primitives

The literature on real-time operating describes many different types of intertask synchronization and communication primitives [SR87][Sha84][PS87][Ber86][Rea86]. While these primitives all have some unique properties, they all perform the same basic functions. These functions can be grouped as follows:

1. SEND a message
2. RECEIVE a message
3. WAIT for synchronization
4. ABORT a message
5. CHANGE communication partners

It is important to note that not all of these functions are required in all message based systems. Only the first two are deemed to be essential while the last three may or may not be required, depending on the nature of the system. The send and receive functions are used for the actual exchange of data. The wait function is
required if the system uses an asynchronous message passing scheme but the tasks still require occasional explicit synchronization. The `abort` function is used to nullify messages that have been sent but not yet accepted. It may be useful in any messaging environment. Finally, the `change` function is used in systems that employ semi-static links between communicating partners. The `change` function is used to modify these links.

### 3.2.3.1 Send and Receive Operations

In the design of an intertask communication scheme, the `send` and `receive` primitives are so closely tied together that they can not be considered individually. One is complementary to the other. After the issue of using a blocking (synchronous) or nonblocking (asynchronous) protocol has been decided, the required actions of the `send` and `receive` primitives can be defined. A request for a `send` operation must specify a destination task or link (for dynamic addressing only) and a block of data to be transmitted. In the case of a blocking send, the sending task is blocked until the sent message is accepted by the receiving task or until the receiving task explicitly frees the sender through some type of `reply` primitive. If the `send` is a nonblocking operation, the block must be stored in the designated area (either in kernel space or user space) until the corresponding `receive` operation is performed. A nonblocking environment may use either a blocking or a nonblocking `receive`. In the blocking version, if there is no pending message, the receiving task is blocked until a message is sent to it. In the nonblocking version, the receiver is not blocked if there is no pending message and therefore must poll for incoming messages. If a blocking `receive` primitive is used, the `send` primitive can be made conditional in that it sends its message only if the receiver is already blocked waiting, otherwise it returns a failure indication. This type of conditional send effectively eliminates the need for buffering in an asynchronous environment. If as in this case, details about the success of the send operation are to be made available to the sending task, the `send` primitive must provide a space to store a completion or error code of the send operation when it is completed.
The receive primitive is used to copy data from the sender's buffer (or from a kernel buffer) into the receiver's buffer. If the receiving buffer is too small, the primitive may be required to return a warning to the sender and/or the receiver as the operation completion result. If the system uses dynamic addressing, the receive primitive can be called either with a specific task name (link) or with an any_task (any_link) parameter which allows reception from any source. The receive specific primitive allows discrimination in the kinds of messages that a task is willing to receive at a particular time. However, this capability does pose a problem. The messages from sending tasks that are not currently eligible to be received must either be rejected or queued. If they are queued, the situation may arise where a receiving task is habitually waiting for messages from a particular task while messages from other tasks continue to pile up. This type of situation must be detected in the design phase so that the need for appropriate restructuring and buffer capacities can be identified.

To provide some semblance of flow control in the asynchronous environment, upper bounds must be placed on the message queue sizes or the communicating tasks must be supplied with some additional synchronization primitives. Restricting the sizes of message queues has the effect of complicating the send primitive to handle the full queue situation. For this reason, it is desirable to allow the application tasks to manage their own flow control.

3.2.3.2 The Wait Operation

The wait operation is intended not only as a means of flow control on message production, but also as a means of satisfying precedence constraints between tasks. This type of primitive is necessary in an asynchronous messaging environment. To provide synchronization, the wait primitive must be tied to some type of identifiable action such as the transmission or reception of a message. When used in this manner, wait must be called with a specific task name or link identifier. When the primitive is called, the calling task is to be blocked until a message is received from the specified task or link. It may also be desirable to design the wait primitive such that a sender may block itself until its message is received.
VAR synch ; Synchronization Variable

TASK 1

WHILE (work left to do) DO
  process work
ENDWHILE
WAIT(synch)
Process Fault Condition

TASK 2

Perform Data Calculations
Perform Hardware Access
IF access problem THEN
  CAUSE(synch)
ENDIF
Continue Processing

Figure 3.7: Cause/Wait Synchronization

The separation of the synchronization primitive from the communication primitives provides an added degree of flexibility in that blocking is done only when needed rather than at every communication. Another variation of wait removes communication from synchronization altogether. This requires the introduction of signalling variables and a corresponding cause primitive. If two tasks wish to synchronize, one would issue a wait request on a specified signal. The wait primitive would then block that task until some other task performs a cause operation on the same signalling variable. This set of primitives allow synchronization without the added overheads of messaging. Their use is demonstrated in figure 3.7.

3.2.3.3 The Abort Operation

The abort operation is used to cancel messages that have been sent but have not yet been accepted. This may not seem to be a very important function but it may be useful in some cases where time of message response is an important factor. For example, if a task has sent a request message to another task and sufficient time has elapsed such that the response would no longer be useful, the sending task could
call the *abort* primitive to prevent the receiving task from wasting processing time on the now useless response. This course of action is feasible only in a nonblocking environment. If the *abort* primitive is made unrestricted in allowing any task to cancel any pending message in the system, the *abort* primitive could also be useful in a blocking environment that contains watchdog tasks. If the watchdog tasks are able to detect deadlock situations, they could relieve the deadlock by preempting messages that have not been received and thereby freeing the sender. The *abort* primitive is nonblocking in all cases.

3.2.3.4 The Change Operation

The *change* operation is necessary when dynamic links are used in the system to explicitly specify communication partners at some particular instance in time. This might be necessary if the communication needs of the system require occasional changing of the communication partners or if the system has dynamic task creation and deletion where task names can not be explicitly specified. When a request to change a link is issued, the kernel may be required to inform another node of the change request if the change involves partners on different processors. After any pending communications have been completed, the receiving kernel (if appropriate) must acknowledge the change request. The calling task is blocked during this time (until the acknowledgement is received) to prevent further send operations on the link. Of course this method eliminates only one direction of communication. A decision must be made on when the link may actually be freed. This can occur after one of several possibilities:

1. Immediately after one partner requests the change,

2. After the other partner is notified that the link was changed at the request of task X,

3. After all of the pending messages buffered on the link have all been received,

4. Only after both partners have requested a change operation on the link.
The first two of these do not allow the receiving task to examine any messages that may be queued waiting to be received. Also in approaches 1 and 3, if the nonchanging task attempts to perform an operation on a link to which it is no longer connected, it will get an error even though it may not be responsible. The last approach avoids this scenario and ensures that errors depend on a task's own behavior and not on circumstances beyond its control. However it does incur an overhead in the operating system as it would be necessary to keep track of all pending change operations on each link.

This section has shown how various alternative IPC schemes can be used in the design of a multitasking system. The manner in which the communication and synchronization scheme is designed has a large bearing on the structure and partitioning of the application software that will eventually use the facilities.
3.3 System Partitioning

System partitioning in a multitasking system involves two aspects: assigning operating system functions to the individual processors and grouping application activities into tasks as well as assigning them to available processors. Often, these aspects are glossed over in system design but it is a very important aspect in determining the overall efficiency of the system. Improper partitioning can create bottlenecks both at the interprocessor communication medium and at the interfaces of the application tasks.

Proper system partitioning helps to reduce the amount of unnecessary communication across processors and between tasks on the same processor. At the same time, the partitioning process must avoid restricting system concurrency. It is necessary to strike a balance between these two often conflicting requirements. The partitioning process consists of determining how the operating system functions are to be distributed across processors, how the different activities are to be grouped into tasks, and how the tasks are to be distributed among the processors.

3.3.1 Distribution of Operating System Functions

In terms of the operating system, it is possible to have varying degrees of distribution in a multiple processor environment. At one extreme is the master/slave implementation in which a single master operating system oversees the functions of slave kernels running in each application processor. The opposite extreme is characterized by a fully distributed operating system in which each processor runs a complete kernel which is independent of all the other kernels. In practice, the second extreme is quite difficult to achieve in that most implementations require at least some type of centralized control in the form of global task name tables, or central task scheduling, for example.

In determining what degree of autonomy to grant to each processor's kernel, several factors must be considered. A more distributed kernel is less prone to failure because of the lack of centralized services. Required services are available locally so
local processes are not dependent on remote processors for system services. However this may be an academic consideration as the local processes may still require the services of the applications running on the disabled processor. As this application coupling is inherent in embedded multiple processor systems, the distribution of system functions is not important for reliability. However, distribution does have significant implications on system performance.

Some of the literature has advocated dedicating separate system processors to performing some of the operating system functions that are common to all application processors. These include the global scheduling of responses and communication between tasks [SR87][AA82]. This increases the performance of the application processors as they are now relieved from performing system duties. In addition, this centralized approach allows performance enhancing features such as dynamic load balancing and global task scheduling to be performed. These features would incur unacceptable communication costs if they were distributed across several kernels. Also, because these features are rather computationally expensive, they would be infeasible if they shared processing resources with application software.

An undesirable consequence of the centralized approach is that it tends to increase the complexity of the operating system and during times of heavy load, it may create a bottleneck. If one single processor is now responsible for all system scheduling, interprocessor communication, and the initial stages of interrupt handling, it will very quickly be overrun in larger applications. Depending on the nature of the interactions within the application, the solution may be to centralize some of the common kernel functions such as dynamic load balancing in a dedicated system processor and to duplicate the other oft-used kernel functions such as local node scheduling and the handling of intertask communication. This approach is illustrated in figure 3.8. This design requires a copy of the kernel running in each processor with communication links between the kernel copies. Intertask messages across different nodes require the assistance of both kernels whereas an IPC message within a single node needs only a single kernel. In neither case would the system processor be involved.

The design phase decisions regarding the partitioning of the operating system
Without Duplication of Kernel Functions

With Duplication of Kernel Functions

Figure 3.8: Effect of Duplication of Kernel Functions
functions can be eased again through the use of a design prototyping tool. The prototyping tool could be used to determine the point at which an application becomes too complex to use a centralized kernel. If this is deemed to be the case, the prototype can assist in deciding which kernel services are used often enough by the application to warrant duplication on each processor. Similarly, decisions could be made regarding which sets of nodes require direct interprocessor connections for communication. The assistance provided by the design prototyping method is not limited only to operating system partitioning. It also extends to the partitioning of the application software as well.

3.3.2 Application Software Partitioning

Most real-time applications can be broken down into a number of activities to be executed in required precedence relations. Ideally these activities can be identified and grouped into logical tasks which can run concurrently with one another. Each task may control data or resources of which it is the primary user. However, it will likely also need to use data and/or resources that lie in the domain of other tasks. It is this need for outside data and resources that gives rise to interactions with other tasks. The nature and the number of these interactions is determined by how the application is partitioned. At one extreme, the entire application could be designed as a single task. This would alleviate the problem of the task needing to access data or resources outside its domain, but it would also limit performance by eliminating any concurrency. Even at the other extreme (one task for each activity), concurrency would be limited by each of the numerous tasks colliding in their external data/resource requests. The actual partitioning must be a compromise between these two extremes.

Today, application partitioning is often done without a complete analysis of the system's constituent activities and is based only on resource usage. Without this type of analysis, it is possible to run into problems such as overconstraining the order of activity execution and thereby prohibiting parallelism through subroutining [Gen81]. Subroutining occurs in a blocking environment in which one task sends
a request message to a second task and is blocked until the request is completely finished. The same execution effect would occur if the sending task had simply called a subroutine to perform the required request operations. In such cases, the programs execute correctly but unnecessarily inefficiently. This type of design error tends to occur when different activities are sequentially dependent on one another but the application has been partitioned in such a way as to allow them to run in parallel. The added concurrency in this case provides no performance advantage. Instead, it often tends to degrade overall performance by unnecessarily increasing the number of tasks in the system. Similar performance errors can be found in the design of nonblocking systems as evidenced by long pending message queues full of unserviced requests. To better distribute control over resources and to prevent such errors, the client-server model is extensively used.

3.3.2.1 The Client-Server Paradigm

A systematic approach to the partitioning of an application involves designating tasks to perform a particular service or to oversee the usage of a particular resource. This serves to control access to resources by encapsulating them within individual tasks. Thus individual application tasks become associated with particular services or resources. This allows all of the work to be done by the application to be specified explicitly in terms of the messages that are passed between the tasks. This leads to a server-oriented [Gen81] design viewpoint.

The application software thus becomes data driven at the task level with activities being performed based on the messages being passed in the system. Mutual exclusion of shared resources or data follows directly from the server task having only a single thread of control. Synchronization follows from the server having complete control over the time at which it chooses to service a particular request. Concurrency is introduced by using nonblocking primitives and allowing the server to examine more than one work request at a time.

In this client-server model, client tasks can perform operations on resources or access shared data by sending messages to the server requesting it to do the operation
on their behalf. This approach is shown in figure 3.9 using Buhr's task notation [Buh84]. The client's message contains information about the details of the request such as the requested operation and perhaps a deadline by which the request should be completed (or its relative urgency) if it is known. For an often used critical service, it may be necessary to provide a number of duplicate server tasks. If this is the case, a mailing system in which request messages are sent to a single intermediate buffer or mailbox from which they can be accessed by the first available server task may prove useful as shown in figure 3.10.

In these terms, the entire application can be viewed as a series of servers and clients requesting operations from each other to provide the required responses. Although it is fairly straightforward, the basic client-server partitioning model does have its limitations. If a blocking message passing scheme is used, concurrency is limited as the requestor is blocked until the request (or critical portion thereof) has been serviced. This limitation can at times leave high priority requests waiting for previously sent lower priority requests to be serviced. In a real-time system, this situation is unacceptable. The situation can partially be remedied by assigning priorities to queued request messages but this still does not allow a high priority request to start
Figure 3.10: Mailbox Effects on Client-Server Interactions

While a lower priority one is in the process of being serviced. Thus the main problem with the model is that it handles requests only in the order that they arrive and that it can handle only one request at a time.

In hopes of remedying this situation, Gentleman has introduced a variation of the client-server model known as the administrator which oversees a number of worker tasks (as shown in figure 3.11). The worker tasks are used to increase concurrency by handling multiple requests to the administrator in parallel. Even in a blocking message environment, the administrator task is designed so that it is never blocked. Rather it is the worker tasks that are blocked waiting for work assignments. This guarantees that requests will be serviced as soon as they arrive provided that enough worker tasks are available. If all of the worker tasks are busy, the administrator must again queue further work requests until a worker task is available. Mutual exclusion is guaranteed because the administrator does not release workers unless their activities are disjoint.

In deciding on the best way to partition an application, a tradeoff may have to be made between the client-server and administrator models. Since an administrator hides its workers, it also hides the number of instances of each kind of worker. In some situations, there could be a very large number of worker tasks, most of which
would be blocked most of the time waiting for something to do. So while the administrator increases system concurrency, it also creates large numbers of tasks which serve to increase system overhead in terms of space limitations and queue and task management.

A design prototyping tool is extremely useful in determining when a single server type task is not good enough to achieve required performance. In this case, it could ascertain how many duplicate servers are required to adequately service the needs of the application or when the administrator concept should be used to clear a system bottleneck. If the administrator is deemed to be necessary, the tool can assist in determining the optimal number of worker tasks that should be used to maintain adequate parallelism and concurrency for the expected load conditions. In addition, the design tool can be used to catch such performance bugs such as subroutining which can arise in the application partitioning process.

3.4 Event Handling and Response Scheduling

The final aspect of the design phase in which a design prototyping tool is essential is in the making of decisions regarding the type of task scheduling and event handling that is best suited for a particular application. In making this type of decision, several
factors must be considered. This section introduces some of these factors but they are treated more in depth in the context of applying design tools to their resolution in the following chapters.

An embedded system must deal with different types of environmental events. Some may be periodic and some may be sporadic. Each such event triggers a series of actions which are typically implemented within a set of application tasks. The actions of these tasks ultimately lead to a response to the environment. The environmental events may occur asynchronously and the internal actions triggered by these events will generally vary depending on the current state of the system. For this reason, it is necessary to implement some form of task scheduling scheme that can properly sequence the internal actions based on immediate system requirements.

3.4.1 Initial Event Handling

The first phase of initiating a response to an environmental event is the handling of the event itself. Typically, real-time systems are characterized by a wide range of I/O devices used to convey information between the environment and the controller. Because of this large variety, it is usually the application tasks that are responsible for servicing a particular device. The interactions between these devices and the controller are usually implemented by way of processor interrupts and in general, it is impossible to schedule a servicing task to run immediately when its device interrupt occurs. There is a time overhead associated with performing a context switch which may result in the loss of data associated with the interrupt. In addition, the scheduling algorithm may not see fit to preempt the currently executing task in favor of the service task. For this reason, event occurrences are always recorded first by interrupt handlers, whose primary responsibility is to ensure that data is not lost.

To achieve device independence, these handlers are implemented in the application software and buffers are provided between the tasks and the interrupt handlers for temporary event data storage. If the interrupt calls for further servicing by an application task, the handler signals the scheduler of the event occurrence. At this point a scheduling decision may be made regarding when the application task that is
principally responsible for initiating the response should be run. Thus the interrupt
handlers may preempt a task but the preemption is transparent to the task as the
handler execution times are short and they do not modify any relevant data.

3.4.2 Difficulties in Real-time Scheduling

Each application task can be characterized by the processor time it requires to com-
plete its operations and also by the hardware and software resources that it requires.
The amount of time required by a task for completion is a very difficult quantity to
accurately predict. Each task may provide one of a number of different actions each of
which may take a different amount of time. In addition, each of these actions can have
timing variations in the form of communication delays, DMA cycle stealing, operating
system overhead, bus contention, and the time required for actuators, motors, and
other physical devices to work must all be considered where applicable in determining
the exact timing requirements. Since this is not feasible in a dynamic environment, an
estimate of the worst case task time is one measure used to characterize the task. The
hardware requirements of the task refer to the specialized (possibly shared) resources
that it may require in the course of its execution. Similarly, its software requirements
refer to the task's need for data or services provided by other tasks in the system.

The scheduler therefore must be able to deal with many nondeterministic factors
including task execution time and the possibility of the occurrences of more critical
events requiring high priority servicing. If there are significant differences in the ur-
gencies of different responses or if the response deadlines have a very limited degree of
laxity, real-time scheduling schemes must implement some type of resource and pro-
cessor preemption so that high priority tasks are not kept waiting for lower priority
tasks that are currently executing. These preemption schemes must be implemented
in a manner that not only considers the immediate requirement of servicing the high
priority task but also the longer term impacts of the required preemption. For exa-
ample, in the case of an I/O channel, if a low priority task is preempted during the late
stages of data transmission, the entire message will have to be retransmitted at some
later time. This incurs additional overhead in the retransmission of the already sent
bytes. In the case of this resource preemption, in the overall performance picture it may be desirable to let the current lower priority user complete its operation rather than immediately preempting the resource in favor of the high priority user.

The overall efficiency of the system is determined to a large extent by the way in which priorities are assigned to the tasks in the system. In many cases, priorities are assigned based on the perceived importance of a particular function. In fact, many additional factors must be weighed in assessing task priorities. A service task that performs some background operation may not be considered to be very important but it may warrant a high priority if it is used by many client tasks and could therefore cause a potential bottleneck. Similarly one must be careful in assigning a high priority to a relatively important operation that takes a very long time to complete. In a static priority system, this long term task may starve smaller slightly lower priority tasks.

3.4.3 Static and Dynamic Scheduling

The perceived importance of a particular operation may not be constant in time. The priority may have to be adjusted dynamically to account for changing system conditions. This complicating factor has given rise to a variety of dynamic scheduling algorithms which are used to accommodate changing conditions by changing the order in which tasks are scheduled to run. In contrast, static scheduling schemes simply base all execution schedules on task priorities that are constant throughout the life of the system.

Because of the typically large number of tasks involved in an application (large combinations of which may be active at a given time) and the continually changing demands on the system, it is generally impossible to determine static priorities for each task that guarantee the required system responsiveness. This is the rationale behind using dynamic scheduling. However, while dynamic scheduling may provide better adaptations to changing system conditions, it does come with a time overhead. Because static scheduling simply arranges tasks ready for execution in order of their priorities, there is very little computation time involved in scheduling. Dynamic scheduling however may have to compute things such as least slack time, which task
will complete the fastest, etc. depending on the dynamic scheduling algorithm to be used.

The scheduling policy of the SPRING kernel [SR87] even takes task resource requirements into account and attempts to avoid conflicts by identifying tasks which contend for exclusive use of a given resource and scheduling them for different times. While this type of scheduling saves time by reducing the number of context switches and by reducing the amount of idle processor time, the extra computations take processor time away from the application tasks. If there are a very large number of tasks in the system, the added overhead may become significant, depending on the computational complexity of the algorithm.

The added overhead may be acceptable if the operating system has been partitioned in such a way that the dynamic scheduling algorithm is run on dedicated operating system processors. This approach has been required in the SPRING kernel in which the scheduling functions have been separated from the application. In this system, each application processor is paired with a system processor in which a local scheduler continuously updates the execution schedule for its application node. In addition, there is also a global scheduler which can schedule tasks to be run on any of the local nodes. A task that can not be adequately scheduled by a local scheduler is passed to the global scheduler in the hopes that it can find another node in the system to run the task. For this type of scheme to be feasible, the system configuration must be such that tasks are duplicated across processors.

This brief introduction to response scheduling in a real-time environment serves to illustrate the complexities that must be faced in this part of the design phase. Analytical modelling of the scheduling problem does serve to provide some insight into which approach should be implemented but an approach more customized to a particular application is what is really needed. The concrete design aspects of scheduling involve assigning actual priorities to particular tasks; determining the effectiveness of particular static and dynamic scheduling schemes for the application at hand; determining if static scheduling is sufficient for all expected load conditions or the whether the added overhead of dynamic scheduling will be worthwhile. These
types of decisions can only be made after the behavior of the application in question is observed and analyzed under each of the proposed schemes using an appropriate design phase prototyping tool. While current scheduling schemes consider only the scheduling of individual tasks, what is really needed is a scheduling policy based on complete responses.

This chapter has served to describe the nature of real-time multitasking systems by emphasizing the close ties between hardware configuration, intertask communication schemes, system partitioning, and scheduling. It has also defined some of the problems inherent in the detailed design of these systems. The next chapter outlines the specific design phase decisions that must be made in each of the areas of the multitasking platform in order to come up with a working design. In addition, it specifies the type of statistical factors that must be considered in making these decisions. These are the first steps in defining a systematic design methodology.
Chapter 4

System Design Aspects

The following three chapters combine to provide a methodology that allows designers to systematically analyze a proposed system specification and make design level decisions based on the analysis. At this stage, it is presumed that the functional requirements specification phase has been completed and the design phase of the development cycle is about to be entered.

This chapter begins the process by summarizing the various design decisions that need to be made at this stage and then discussing the types of data and statistics required to make these decisions. This is followed in the next chapter by the introduction of the Multitasking Design Assist (MDA) simulator that is instrumental in modelling the effects of these design decisions by gathering the data and statistics described here.

4.1 Design Phase Decisions

At this stage, it should be quite clear that there are many ways in which to take a functional specification and transform it into a viable design. The choices a designer makes in realizing such a design are strongly influenced by what he or she perceives to be its ultimate goals. These goals or objectives do not necessarily refer to the system meeting its functional requirements, but more to the way in which it meets these requirements. For example, two systems may be designed in such a way that
they both meet their functional objectives but one may be deemed better than the other because it is more reliable, more easily modifiable, uses fewer processors, etc.

Many of these goals and objectives are common to different types of real-time systems as they all share a need for responsiveness and reliability. To this end, the goals of the design phase of the development cycle can be summarized as follows:

1. Attempt to ensure system responsiveness in that responses are delivered at appropriate or acceptable times (meet all timing constraints),

2. Reduce the amount of information that must be passed between the processors in a multiple processor system,

3. Reduce or eliminate bottlenecks by identifying them and providing duplicate resources for their resolution,

4. Reduce the amount of time that application tasks spend in the ready to run state for lack of a processing resource,

5. Maximize processor utilization by the application software by reducing the amount of operating system overhead,

6. Provide some degree of reliability in the face of processor failure by being able to predetermine and perhaps compensate for the effects of such failures.

The degree to which these goals are achieved depends heavily on the decisions made regarding software partitioning, choice of hardware system, choice of IPC schemes, and choice of scheduling schemes. It should be noted that these four areas are extremely interdependent and that the choices made in any one of these areas will have significant effects on the others. As such, these choices must be based not on instinct or feeling, but on the careful analysis of specific system characteristics and on how these characteristics are affected by a particular decision. The decisions to be made at the design stage are summarized in figure 4.1 and detailed in the following sections.
Figure 4.1: Design Phase Alternatives
4.1.1 Impact of Application Software Partitioning

As indicated in the previous chapter, software partitioning involves partitioning the application into tasks and distributing system software among processors. This latter aspect has been discussed in the previous chapter and this section focuses on application software partitioning. The initial partitioning of application software is the most fundamental of the four classes of design decisions as it has a direct impact on the other three. Software partitioning refers to grouping low level application activities into tasks. Improper system partitioning results in many types of performance degradations. These are described in the following list.

1. One of the first decisions to be made deals with task granularity. Having tasks with too fine a granularity (few activities and/or very small activities) serves to increase the potential concurrency in the system but it also leads to a large number of tasks. The greater the number of tasks, the greater is the operating system overhead incurred in their management. This overhead takes the form of memory consumption and time usage. Every task instance requires a fixed amount of memory for its task control block as well as for its IPC structures. In addition, if there are a large number of tasks in the system and each task is responsible for only a few activities, it follows that a large number of context switches will be required to do the complete job. This may be an acceptable price to pay for the added concurrency if the chosen processor has special mechanisms to reduce the time involved in a context switch.

2. Having many fine granularity tasks does not in itself guarantee a high level of concurrency. If the system is partitioned without paying proper attention to the precedence relationships between activities, a situation may arise in which in addition to the large overhead due to too many tasks, concurrency is also restricted because activities that are sequentially related to each other have been grouped into separate concurrent tasks. This kind of partitioning leads to subroutining.
3. Failure to group activities according to their data dependencies will also have a detrimental effect on system performance. In this case, if activities that need to exchange large amounts of data are grouped into separate tasks, system overhead is increased as a result of the operating system having to handle larger volumes of intertask data. If the activities are grouped into tasks on different processors, the problem is compounded due to the increased usage of the transmission medium or the increased shared memory contention.

4. Failure to group activities that share the same data or resources into separate tasks has the effect of increasing the possibility of collision over the data or resource and thereby resulting in a reduction of overall system concurrency.

5. If the tasks have too coarse a granularity (contain many activities and/or large activities), the number of tasks in the system is reduced but the exploitation of possible concurrencies may also be reduced. This concentration of function may result in bottlenecks. In this situation, a particular task may be unable to respond to service requests or external events at a fast enough rate due to the large number of sequential activities that it must perform. This leaves other tasks waiting for the completion of the bottleneck task. This situation also arises if there is a large mismatch in the execution times of interacting tasks.

6. In addition to task granularity and data dependencies between activities, task functionality must be considered in the partitioning process. If one type of functionality is encapsulated into a single task, the system becomes vulnerable to failure. This is particularly true in the case of service tasks which may be required to support several different parts of the system.

These undesirable situations can be avoided by careful consideration of the characteristics of the atomic activities and by determining how they interact with each other. Once the activities have been initially grouped into tasks, the tasks must be ported to a chosen hardware platform.
4.1.2 Impact of Hardware Platform

The hardware platform includes the number and choice of processors used to implement the system as well as the scheme used to interlink the processors. A discussion on the choice of microprocessors used to implement a particular system is beyond the scope of this thesis as this type of decision is part of the implementation phase and is more of a consequence of the design phase decisions. Occasional reference is made only to general processor features such as processor capabilities (speed, the availability of register windowing or special instructions for fast context switches, etc.) which may influence some of the decisions made in the design process. Instead, this discussion focuses on the two remaining components of the hardware platform.

4.1.2.1 Number of Processors

Choosing the correct number of processors for an application is influenced by a number of factors. Many of the arguments relating to increasing concurrency through large numbers of fine granularity tasks can also be applied to increasing concurrency by adding additional processors. Depending on the nature of the task interactions, it is true that adding additional processors can increase system throughput. However, this must be balanced against the additional overheads incurred with greater numbers of processors. Let us treat each of these aspects individually.

In any multitasking system, it is necessary to have some interactions between the individual tasks. These interactions can be in the form of precedence relations between the tasks, communication through messages, access of common data, or the sharing of common hardware resources. Because there are these interdependencies between tasks and because there may be very large numbers of tasks in a system, it is necessary to perform some type of task to processor allocation (assuming there is no dynamic tasking). If this allocation is done properly, it also determines the number of processors required.

There are two main considerations in assigning tasks to processors: alleviation of contention, and reliability. Contention occurs when one or more tasks are prevented from running because the contended item is in use by another task. Contention comes
in many forms including contention for processor time, other hardware resources, data, and service task resources.

Contention for processor time manifests itself in the form of long ready to run queues in which several tasks are logically able to proceed but they must wait for the processor to become free. Improper task to processor allocation can create situations in which some processors have a large number of tasks that are ready to run while other processors are idling because all of their tasks are blocked waiting for something other than the processor. This is a case of improper load balancing which can be avoided by assigning interdependent tasks to the same processor as shown in figure 4.2. In this case typically only a single task in the dependency set is able to run while the other tasks in the set are waiting not for the processor, but for the currently executing task.

In many cases, it may be difficult to alleviate contention for hardware resources other than the processor (buses, memory, and I/O devices) through different task allocation schemes. Such resources are typically associated with one or more processors in the system. Some of these resources are accessible only with specific processors in the system or are less time consuming to access from these processors. As such, all tasks that need to use these resources must be resident on the processor associated with the resource. If all these tasks are concentrated in this manner, a large amount of contention is inevitable. The only way in which to reduce this type of contention is to duplicate the resource across several processors.
In order to reduce the amount of contention for shared memory or for the interprocessor communication medium, tasks that use the same data should be resident on the same processor. Finally, a similar situation occurs in the case of service or server tasks. If these tasks do not have specific hardware requirements, it is necessary to allocate them to the same processor on which the majority of their client requests originate. This not only serves to reduce the time required to service the request, it also decreases the volume of interprocessor traffic.

After an initial task to processor allocation has been done, an assessment can be made regarding the number and types of processors needed to provide adequate system responsiveness. In some cases, even after appropriate load balancing has been done, several processors may still consistently have large numbers of ready to run tasks. This is the main situation that warrants the introduction of additional processors into the system. As additional processors are introduced, the task to processor allocation procedure must be repeated to ensure proper usage of the new resources. This cycle of processor additions and task to processor allocation continues until adequate event response times are achieved.

The other main reason for adding processors to a system is the need for reliability. Ideally, the system is designed so that a single processor failure has only a localized effect on system performance. Various schemes can be employed to reach this goal but most of these rely on some type of duplication of functionality. This duplication applies to both software and hardware and can be configured either in a standby mode, a load sharing mode, or some some variation of these modes. In the standby mode, duplicate processors contain the same tasks and the additional nodes are inactive as long the primary processor is functioning properly. In load sharing, tasks are duplicated on different processors and both the primary and the backup node are active but in case of failure, the remaining node is capable of handling the workload of both nodes. In any of these cases, the main concern of the designer is to be able to identify the consequences of a processor failure. Once the vulnerable functionalities are identified, measures such as adding backup processors can be taken.

If the determination of the number of processors required for a particular system
is not arrived at systematically, it is probable that extra processors will be needed
to compensate for the improper usage of the existing processors. However, just like
additional tasks, the inclusion of additional processors comes at a cost and must be
done only when necessary. As the number of processors in the system increases, OS
functions such as system-wide scheduling or arbitration of shared data and resources
become more time consuming. Also as more processors are added, there is an increase
in contention for shared memory or for the interconnection medium between them.
The degree of this degradation depends on the amount of interdependence between
the software tasks running on each processor and on the algorithmic complexities
of the corresponding software functions. At some point, the added concurrency of
the additional processors is nullified by the resulting slowdown due to the added
contention.

4.1.2.2 Processor Interlink Scheme

Once the number of processors required for a particular application has been deter-
mined, the other major hardware decision lies in selecting an appropriate way to link
the individual nodes. The two extremes of this decision are to have a single global
memory shared by all processors or to have no shared memory but rather to connect
all the nodes through some communications system.

The first step in this procedure must be to identify which of the processors in
the system are required to communicate with which other processors. Once this has
been established, several factors must be considered and weighed against each other
in arriving at a final scheme. The communication pattern might be such that several
clusters of nodes can be identified in which there is a high level of communication
activity within the clusters but much less between them. Or it may be that the
level of activity is constant between all nodes in the system. The first situation might
suggest using a heterogeneous linking scheme in which communication within a cluster
is carried out using shared memory while intercluster communication is done through
a communications system, or vice versa depending on the amounts of data that need
to be exchanged. In contrast, the second situation might suggest a homogeneous
approach in which all communication is done via a global shared memory or via a
fully interconnected communications system.

In fact it is impossible to predict what is the best method by this type of super-
ficial analysis. The only way to gain reasonable certainty that a chosen method is
appropriate for a particular application is to model it and observe its effects on the
system. The consequences of the chosen interlink scheme must be considered in light
of the effects they will have on the overall responsiveness of the system. If memory
is shared between too many nodes, the delays incurred due to bus contention may
be substantial. In addition, if writes to the shared memory have to be echoed to
the individual processors' private data areas, overhead is again increased. A partial
solution to these problems may be to divide the shared memory into several local
memories. The processor that makes the most frequent accesses to a local memory
will access it directly, relegating the other processors to access through some kind
of arbitrated access such as cycle stealing. The success of such compromises can be
measured by modelling the changes in the system and analyzing their results.

An analogous situation arises in the case of the communications system. Rather
than incurring the costs of a fully interconnected system with links between all pro-
cessors, the model should be analyzed to determine where traffic volumes justify a
communications link. However, it is not only the nature of the interprocessor inter-
actions that must be considered in deciding upon an appropriate interlink scheme.
The operating system aspects of the IPC primitives must also be considered.

4.1.3 Impact of Intertask Communications

In many cases the choice of IPC and other operating system primitives is not up to the
system designer. Very often, the new design is simply an add-on of new features to a
larger existing system. In such cases, relatively few decisions have to be made about
IPC mechanisms. However, in some cases the large project may be just beginning or
a small standalone system may need to be designed from the start. In these cases,
an examination of possible IPC schemes is warranted.

This examination consists of three main aspects: the identification of the needs
for the various IPC primitives, the costs of using these primitives, and the influence these primitives have on the structure of the application software.

4.1.3.1 Types of IPC Primitives

There are essentially two types of communications in a multiple processor system: those involving tasks on the same processor and those involving tasks on separate processors. To maintain consistency, it is desirable to have a uniform interface for either type of communication. For this reason, the primitives are chosen to reflect the nature of the underlying interlinking scheme between processors.

If this scheme implements a loosely coupled system, then the choice of primitives is limited to some variation of the send and receive message operations. Additional flexibility is possible if the interlinking mechanism is a shared memory. In shared memory systems, two tasks communicate through the use of global data. Because the global data accesses are asynchronous, the IPC primitives in these systems serve mainly to arbitrate and avoid conflicts over the shared resource. An additional message layer can also be superimposed on top of the data arbitration constructs. This additional layer serves only to provide a more natural syntactic IPC interface to the application tasks. IPC involving tasks on the same processor is very similar to inter-processor communication using a shared memory. In essence, the private processor memory is shared between the various tasks in that processor.

Thus it follows that messaging constructs such as send and receive may be implemented either on shared memory systems or on loosely coupled systems. In contrast, shared memory constructs such as semaphores are easily implemented only on shared memory systems or within the individual processors of loosely coupled systems. Hence the choice of which class of IPC operations the system should support hinges to a great extent on the choice of processor interlinking scheme. Because of their general application, the remainder of this discussion will focus on messaging IPC constructs.

The major decision in terms of the send and receive primitives involves the selection of a blocking or nonblocking protocol. A nonblocking protocol may help in increasing the amount of concurrency possible in the system but it may also incur
problems in flow control with one task flooding another with service request messages. As well, it also imposes the need for message queue management. A blocking protocol on the other hand automatically takes care of the flow control problem. The final decision as to which protocol best meets the needs of the system can be made by prototyping each of the alternatives and examining what differing effect (if any) each has on the system.

Apart from the send and receive functions, specific applications may have requirements for specific primitives. An abort type primitive serves to eliminate messages that have been sent but not yet received. This primitive could be useful in nonblocking systems in which unserviced request messages that have missed their deadlines could be removed from the system by the requesting task. In dynamic tasking systems, other primitives such as create and destroy may need to be considered. The decision on whether to include such primitives must be made after the performance benefits gained by its inclusion have been analyzed.

4.1.3.2 IPC Costs

If the operating system is to be designed along with the application, the costs associated with various IPC mechanisms must be considered. IPC costs include things such as: the number of copy operations associated with a send/receive primitive, the amount of memory and time overhead needed for maintaining message queues, the costs associated with saving and restoring contexts during context switches, and the number of addressing identifiers that will be needed for the system.

While these costs of time and memory space may seem to be inconsequential, their cumulative effect when measured over many repeated invocations, may be significant. Again, prototyping represents a viable method by which to quantify such overheads.

4.1.3.3 IPC Effects on Application Software

The choice of messaging primitives included in the system has a direct effect on several parameters in the realm of the application software. These effects deal mainly with the required complexity of the application software and with message buffer sizes
and flow control in the asynchronous environment. If the OS provides only "bare-bones" support for IPC, functions such as queue management may be shifted to the application software. In any case, if nonblocking messaging is used, consideration must be paid to allocating the proper amount of buffer space so that all anticipated rates of message transmission can be handled.

This point is especially important in the case of interprocessor messages. For these messages, each processor's kernel must buffer them as they are in the process of being received. In order to prevent all buffers from being in use while messages are in the process of being received from several sources, the kernel must be allocated an adequate number of buffers.

In a similar vein, intertask messages that have been sent but not yet received must be buffered. The choice of an appropriate buffer size for each task is especially important in systems that do not support dynamic memory allocation. In such static systems, overestimating the buffer requirements results in wasted memory while underestimates result in the loss or delay of message transmission.

If the application software is based on the client-server model, one of the objectives of intertask communication is to allow client tasks to delegate work to server tasks. Since server tasks typically handle requests from many client tasks, their maximum queue sizes must be chosen to be able to accommodate peak request instances. In these cases, the sizes of the server message queues can also be an indication as to the number of server tasks required to keep the request backlogs at acceptable levels. Decisions must also be made on the number of required servers and on how to allocate the requests among the available server tasks.

Since the client-server model allows different tasks to make use of common servers, the situation may arise in which the requests of urgent tasks are delayed because of an influx of requests from less urgent tasks. This situation can be alleviated by providing the server task(s) with multiple message queues or by associating priorities with messages. System prototypes can indicate whether a single mailbox messaging approach provides better responsiveness than associating a message queue with each server task. The decision on whether these types of modifications to the messaging
system are warranted can again be made by examining system behavior through prototypes.

4.1.4 Impact of Scheduling

The attributes of the scheduling scheme chosen for a particular system have a major impact on the effectiveness of the controller in providing timely responses to external events. The decisions that need to be made concerning scheduling can be grouped into four main categories: whether to use dynamic or static scheduling, how to assign task priorities, whether to use preemptive or nonpreemptive scheduling, and whether to use localized or global scheduling.

4.1.4.1 Static or Dynamic Scheduling

The major driving force behind the use of static scheduling in real-time systems is that it has a very low overhead. In making the decision between static and dynamic scheduling, one must consider the situations under which dynamic scheduling may improve responsiveness. If the system is liable to undergo wide fluctuations in load conditions then the added cost of dynamic scheduling may be compensated by its performance benefits. Also, dynamic scheduling should not be considered as a single type of scheduling. There are many dynamic scheduling algorithms and each has its own complexity and its own optimal working parameters. The algorithm that works best for a particular application under its range of load conditions and processor configuration can be determined by observing its effects on a system prototype.

The costs associated with dynamic scheduling may be overcome somewhat by transferring the scheduling functions to one or more dedicated system processors. In this way, processing capacity is not taken away from the application tasks by the scheduling algorithm. However, this will increase the amount of interprocessor communications traffic in the system.
4.1.4.2 Assigning Task Priorities

The assignment of priorities is most crucial in static scheduling as there are no algorithms to dynamically compensate for poor judgement in initial prioritization. In static scheduling, there must be some way to quantify the urgency or importance of a particular task in relation to other tasks. As a result, decisions must be made on how to distribute priorities among the tasks.

Task priorities must be assigned by examining factors such as the number of responses with which a particular task is involved, the urgencies of these responses, whether the task is mainly a client task or a server task, and the consequences of assigning a low or high priority to the task. If a task is a server task or if it is a part of many responses, it likely warrants a high priority to prevent bottleneck situations. The consequences of such a bottleneck may be the introduction of significant delays in many responses.

In addition, the role of a particular response must be considered when assigning priorities. For example in a telephone switching system, call processing functionality cannot be sacrificed in favor of routine diagnostic or maintenance actions and so the tasks involved in call processing are assigned a higher priority. Unfortunately, this type of analysis is based purely on subjective measures and as such, this part of the priority decision cannot be aided by statistical analysis.

Nevertheless, apart from functional considerations, the effects of various task priority assignments can be observed through prototypes and modifications can be made until satisfactory performance is achieved.

4.1.4.3 Preemptive or Nonpreemptive Scheduling

The decision on whether or not to use preemptive scheduling depends to a large degree on the nature of the application tasks themselves. Specifically, it depends on the granularity of the tasks and on the time constraints of the responses. If the tasks are of relatively fine granularity, it is likely that their processor requirements will come in the form of short bursts. In such cases, a preemptive scheduling scheme will have little added benefit as the time saved by the preemption will be fairly small.
and will likely be nullified by the time required for the extra context switch. Further, if the time constraints on the individual event responses are not rigid, there may not be any justification for preemptive scheduling.

If the application tasks are of coarse granularity and have relatively large processing requirements between system calls, then a preemptive approach may be a necessity. This preemptive approach may involve either switching tasks whenever a higher priority one becomes ready or using time slicing to switch between tasks at predefined intervals. If time slicing is used, a decision must be made regarding the amount of time to allot to each slice. If conditions justify preemption, the overhead of the extra context switch may be insignificant in comparison with the amount of time that would have been required in allowing the current task to complete. The problem is compounded if one or more responses have rigid deadlines by which they must be completed. The tasks comprising these responses may not be able to afford the delay in allowing the current task to complete.

4.1.4.4 Localized or Global Scheduling

The final major decision with respect to scheduling involves the level of distribution of the scheduling functions. Localized scheduling refers to the situation in which a scheduler is associated with each processor and schedules the tasks only on that processor. In global scheduling, a single scheduler decides which task will be run next on every processor in the system.

In making this design decision, load balancing, task duplication, communications overhead, algorithmic complexity, and reliability must all be considered. The main motivation behind having a global scheduler is that such a scheduler, being aware of the state of all tasks in the system, could increase the responsiveness of the system. The major contributor to this enhanced responsiveness would be a more even load balance across all processors. To achieve this load balance however, would require the duplication of tasks across processors. This duplication increases the memory requirements of the system.

The additional memory requirement is only one of the overheads of centralizing
scheduling functionality. Because tasks are duplicated across various processors, the amount of interprocessor communication increases along with the need to keep data current across all task instances. In addition, the concentration of scheduling in itself increases interprocessor traffic as the scheduling commands must be conveyed to each node from a central source. Another overhead may arise from the nature of the scheduling algorithm itself. A scheduling algorithm with an exponential computational complexity may be acceptable for the small number of tasks associated with a particular node but may become completely unsuitable for managing a large number of system-wide tasks. As such, the benefits gained from better load balancing are surpassed by the processing overhead of the scheduler.

The final factor that must be weighed when decideing upon global scheduling is system reliability. Concentration of system-wide scheduling functionality can lead to complete system failure if the scheduling node is afflicted. In the case of localized scheduling, the damage can be limited only to the afflicted processor by proper task to processor allocation. Thus the benefits of dynamic load balancing on a particular system must be weighed against the possible problems brought on by global scheduling.

It is quite likely that the effectiveness of each of the four major scheduling decisions is different under different situations such as low and high event arrival rates or small and large numbers of tasks. As with all of the described design phase decisions, the correct choices must be determined by observing and statistically analyzing which combinations allow the required responses to be completed by the time they are needed. In order for the observations to be valid, they must be made under all expected running conditions.
4.2 Statistics and Data Interpretation

Many of the design decisions described in the previous section are highly dependent on one another and often times there are conflicting results obtained due to the effects of different decisions. For example, in assigning tasks to processors, task grouping according to functionality tends to reduce interprocessor communications but it also tends to make the system more vulnerable to single processor failure. The merits and drawbacks of each decision must be weighed against each other to arrive at a proper solution. Because the consequences of each decision are usually not obvious and often affect several system factors, it is necessary to devise a yardstick or measure of their "goodness".

The best way to quantify the effects of these design decisions is to identify a number of measurable factors that can be used as statistical indicators. The values that these indicators take on as the result of a particular design decision serve to provide feedback on whether or not the decision leads to a desirable outcome. By examining the design phase decisions and analyzing the common requirements of all real-time systems, a number of indicators that are instrumental in the design phase can be identified.

The significance of each of these factors however varies depending on the characteristics of the intended implementation. The statistics gathered from these factors must be applied after considering particular system characteristics. For example, in processors with register windowing, large number of context switches may be perfectly acceptable due to the reduced switching time provided by the windowing. Similarly, in shared memory systems, decisions that lessen the amount of interprocessor communication may be favored in spite of the consequences of concentrating functionality on a single processor. Different applications and configurations have different factors that they wish to optimize. As such, the purpose of gathering statistics on each of the indicators is not to say that one measure is better than another or that one criteria should be maximized while another is reduced. Instead, the purpose is to present data that can be analyzed in a manner to suit the individual needs of any application and its future implementation.

95
The following sections describe the statistical indicators that must be considered in making decisions relating to each of the four major design phase areas. It becomes evident that a single indicator may be the basis for decisions relating to two or more of these areas. As such, the discussion focuses on the relevance of each factor to a particular area. Figure 4.3 correlates the various statistical factors with the decisions that they influence.

4.2.1 Software Partitioning Factors

In order to analyze the characteristics of different activities and to determine the effects of various activity to task groupings, it is necessary to collect data on a variety of indicators. These are described in the following list.

1. *Data and Precedence Relationships*. These types of relationships between activities help to define the maximum amount of concurrency possible in the application. It involves identifying the activities that need to complete before a particular activity can be started.

2. *Percentage of Possible Parallelism Used*. This measure is calculated by determining how many activities were actually started as soon as their precedence and data constraints allowed, as a percentage of the total number that could have been started. It is used to determine the effectiveness of the activity to task grouping. In practice, this is a very difficult (if not impossible) indicator to measure.

3. *Message Queue Attributes*. This indicator has several uses in determining the effectiveness of activity to task grouping. Large message queues can indicate that too much functionality has been concentrated in one task, thereby making it a bottleneck. It may also show the need for multiple instances of server tasks. If many or all tasks in the system develop large message queues, it may indicate poor grouping in terms of data dependencies. The activities that need data from each other have not been grouped into the same tasks.
<table>
<thead>
<tr>
<th>STATISTICAL GROUP</th>
<th>STATISTICAL COMPONENT</th>
<th>DECISIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Data</td>
<td>OS Overhead</td>
<td>System Partitioning IPC Primitive Costs</td>
</tr>
<tr>
<td></td>
<td>Scheduling Overhead</td>
<td>Task to Processor Allocation Scheduling Scheme</td>
</tr>
<tr>
<td></td>
<td>CPU Idle Time</td>
<td>Activity to Task Assignment Scheduling Scheme</td>
</tr>
<tr>
<td></td>
<td>No. of Context Switches</td>
<td>Activity to Task Assignment Scheduling Scheme</td>
</tr>
<tr>
<td></td>
<td>No. and Types of IPC Calls</td>
<td>IPC Protocol Scheduling Scheme</td>
</tr>
<tr>
<td></td>
<td>Sources of Interprocessor Ms</td>
<td>Task to Processor Allocation Processor Interlink Scheme</td>
</tr>
<tr>
<td>Task Data</td>
<td>Time Spent in System States</td>
<td>Scheduling Scheme Scheduling Priorities</td>
</tr>
<tr>
<td></td>
<td>Time Spent Blocked</td>
<td>Hardware Resource Assignment Task to Processor Allocation Scheduling Priorities</td>
</tr>
<tr>
<td></td>
<td>Task Cycle Time</td>
<td>Activity to Task Assignment Scheduling Scheme</td>
</tr>
<tr>
<td></td>
<td>CPU Usage by Task</td>
<td>Activity to Task Assignment Task to Processor Allocation</td>
</tr>
<tr>
<td></td>
<td>Task Data/Resource Requirements</td>
<td>Activity to Task Assignment Task to Processor Allocation</td>
</tr>
<tr>
<td></td>
<td>No. of Times Preempted</td>
<td>Scheduling Scheme</td>
</tr>
<tr>
<td>Queue Data</td>
<td>Max. and Avg. Queue Lengths</td>
<td>Activity to Task Assignment Task to Processor Allocation Message Buffer Sizes IPC Protocol</td>
</tr>
<tr>
<td></td>
<td>Times Elements Spend in Queue</td>
<td>Prioritized Messaging Scheduling Scheme</td>
</tr>
<tr>
<td>System Data</td>
<td>No. of Shared Memory Conflicts</td>
<td>Task to Processor Allocation Scheduling Scheme</td>
</tr>
<tr>
<td></td>
<td>No. of Writes to Shared Memory</td>
<td>Task to Processor Allocation Hardware Platform</td>
</tr>
<tr>
<td>Event - Response Data</td>
<td>Tasks Associated with Each Event.t</td>
<td>Task to Processor Allocation Scheduling Priorities</td>
</tr>
<tr>
<td></td>
<td>Events Associated with Each Task</td>
<td>Activity to Task Assignment Scheduling Priorities</td>
</tr>
<tr>
<td></td>
<td>Avge Time Needed / Response</td>
<td>All Decisions</td>
</tr>
<tr>
<td></td>
<td>No. &amp; Types of Resp. Completions</td>
<td>All Decisions</td>
</tr>
</tbody>
</table>

Figure 4.3: Statistical Factors
4. *Activity Execution Time*. This data can be used in the activity to task grouping to ensure that no one task becomes a bottleneck by avoiding the situation in which one task is assigned many time consuming activities. This is especially important in the case of nonpreemptive scheduling.

5. *Number of Context Switches*. This is an excellent indicator of the overhead incurred by having large numbers of fine granularity tasks.

6. *Average Number of Ready to Run Tasks*. This indicator shows the effectiveness of having a large number of fine granularity tasks. If the number of ready tasks is consistently large, then there is a possibility for a great deal of concurrency but this concurrency can not be exploited due to the large number of relatively independent tasks competing for the same processor. If all tasks must be resident on the processor in question, the amount of system queueing and context switching overhead can be reduced by combining some activities into larger tasks with little loss in actual concurrency.

7. *Percentage CPU Usage by OS*. As the number of fine granularity tasks grows, this indicator can be used to monitor the corresponding increase in OS overhead in terms of scheduling and queue management.

8. *Response Completion Data*. This set of indicators include data on the number and types of responses completed along with the times at which they completed. These factors provide feedback on the effectiveness of the decisions made regarding software partitioning. Decisions that increase overhead without having positive effects on this indicator must be avoided.

### 4.2.2 Hardware Platform Factors

The decisions regarding the proper task to processor allocation, number of processors, and the interlink scheme used to connect the individual nodes are slightly more involved than the software partitioning decisions. As such, a larger number of factors
must be considered in making this set of decisions. Again, task to processor allocation and determination of the number of required processors will be considered together:

1. *Task Data and Resource Requirements.* Once these factors have been identified, the tasks that use the same data and resources can be allocated to the same processor. Also, the tasks using resources on specific processors can be assigned to those processors.

2. *Internode Messaging Data and Shared Memory Access Conflicts.* These indicators show the individual tasks that are responsible for most of the interprocessor communications traffic. The volume of such traffic can be reduced by assigning these interdependent tasks to the same processor.

3. *Message Queue Attributes.* This indicator identifies the sources of all request messages arriving at a server task. To reduce the amount of interprocessor communication, the server task(s) should be made resident on the node generating the largest number of requests.

4. *Blocked State Data.* This data identifies the reasons that a task was moved to the blocked state. If this data shows many tasks habitually being blocked in attempts to access some hardware resource, it may be necessary to duplicate the resource to alleviate contention.

5. *Task CPU Utilization Time.* This measure shows the processor requirements of each task and can be used as a guide in task to processor allocation. To avoid bottlenecks, tasks that regularly requires large amounts of processing time should not be resident on the same processor.

6. *CPU Idle Time.* If this statistic consistently shows wide variations between processors, it can be used to identify which processors are underused. Hence load balancing can be improved by migrating tasks from busy processors to less busy ones.
7. **Average Number of Ready to Run Tasks.** This statistic gives an indication of the need for additional processors. If this indicator consistently takes on large values, adding additional processors may increase total concurrency.

8. **Percentage CPU Usage by OS and Shared Memory Access Conflicts.** These two indicators provide feedback as to the amount of additional system overhead that is being introduced by adding more processors to the system.

9. **Nonresident Dependent Tasks and Task-Response Mapping.** These two factors can be used in combination to determine the effects of a single processor failure. The first indicator shows which tasks not resident on the failed processor would be affected, whereas the second factor is used to identify the responses that would be affected. This information can be used to determine the manner in which tasks and processors should be duplicated for reliability. For example, it may be desirable to duplicate a processor that contains tasks involved in many different responses to prevent complete system failure.

10. **Response Completion Data.** This set of indicators provides feedback on the effectiveness of the decisions made regarding task to processor allocation and regarding the required number of processors in the system. Once the addition of extra processors has no effect or a detrimental effect on this indicator, the system has too many processors.

The other aspect of the hardware platform that needs to be defined is the way in which the individual nodes are interlinked. Again, several indicators must be considered in making this type of decision:

1. **Internode Messaging Data and Shared Memory Access Conflicts.** These indicators show which of the processors in the system need to directly communicate with which others. This defines the initial interlink topology regardless of the method used in its implementation. The number of shared memory access conflicts also gives an indication as to the amount of bus contention that can be expected in the chosen layout. The internode message data includes statistics
on the rate of message transfer between nodes and thus can give an indication as to the amount of bandwidth required in the communications medium.

2. *Shared Memory Writes.* The number of updates made to data contained in shared memory may be of significance in systems that employ cache memory. If this indicator takes on high values, it may indicate that individual processor caching is of little use in the chosen interlink scheme due to the overheads involved in maintaining cache consistency.

3. *Response Completion Data.* This set of indicators can be used to determine whether or not the different interlink schemes have significant effects on system responsiveness.

### 4.2.3 IPC Factors

Statistical indicators are used for a variety of purposes when applied to intertask communications. They can be used to decide on the required nature of IPC mechanisms in terms of protocol and flow control, to determine the costs of various mechanisms, and to determine the effects of each mechanism on the application. Very often, a single indicator provides insights into several such design decisions.

1. *Message Queue Attributes.* The average and maximum sizes of each task’s message queue assists in making several IPC related decisions. Firstly, the queue sizes provide insight into the amount of memory that should be allocated for the message buffers. Secondly, if many of the message queues routinely become very large, it may signal the need for some type of flow control (perhaps through the use of a blocking message protocol). Thirdly, if there are several common server tasks in the system and they show wide variations in the average number of request messages queued, it may indicate the need for a common mailbox to more evenly distribute the workload. Fourthly, the queue lengths of individual tasks can serve to identify bottlenecks in the system.
2. **Number of Delay Primitive Calls and Number of Context Switches.** A delay type primitive is invoked by tasks that voluntarily suspend themselves because they are unable to proceed at the time of invocation. Overuse of this primitive is an indication of overconcurrency in the system. Tasks are made ready even though they can not logically proceed. This increases the number of context switches and indicates the need for a blocking IPC protocol.

3. **Percentage CPU Usage by OS.** In shared memory systems in which messaging primitives have been overlaid, this indicator gives an idea of the cost of the extra messaging layer. The system can be modelled both with and without the extra layer. The difference in OS CPU usage is a measure of the added overhead. If this indicator is broken down into percentage CPU usage by individual OS components, it could also highlight the need for streamlining the messaging and/or scheduling OS functions.

4. **Interprocessor Messaging Data.** This indicator provides the average and maximum number of messages coming into a node at any one particular time. This information is very useful in determining the number of message buffers that should be allocated to each processor kernel for interprocessor messaging.

5. **Time Spent in Message Queue.** This indicator shows the length of time a message spends in a receiving message queue. If the data show that request messages from high priority tasks are spending long periods of time in message queues because of flooding by messages from lower priority tasks, it may indicate the need for some type of prioritized messaging or the need for multiple receive message queues for specific server tasks.

6. **Response Completion Data.** In addition to quantifying the success or failure of specific IPC related decisions, this set of indicators can signal the need for an ABORT type message primitive. If the data show that a large number of responses are being completed well past their useful deadlines, the inclusion of an abort message primitive may be justified. The abort primitive would reduce time spent on servicing useless requests.
4.2.4 Scheduling Factors

Of all the decisions to be made during the design phase, it is those that relate to the scheduling of tasks and responses that make the greatest direct use of statistical indicators. This is perhaps because the effects of scheduling decisions show up quite clearly and conclusively in the indicators. Because there are many different aspects to scheduling, a number of indicators are required to provide adequate feedback:

1. *Time Spent in Each System State.* The objective of any scheduling scheme is to reduce the amount of time each task spends in the ready to run state. If this is not possible for all tasks, then preference must be given to the more important tasks. This gives an indication of the effectiveness of the task priority assignment scheme. The higher priority tasks should always spend less time in the ready to run state. This indicator must be examined under varying load conditions to get an idea of the merits of the chosen priorities. If the system does not perform very well with changing loads, it may signify the need for a dynamic scheduling algorithm.

2. *Percentage CPU Usage by OS.* This indicator can be used to measure the costs of different scheduling schemes and specifically to determine the amount of overhead associated with a dynamic scheduling algorithm. This is done by observing the difference in this statistic under static and various dynamic scheduling schemes. To get a clear idea of the computational complexity of a particular dynamic scheduling algorithm, this statistic must be taken over a wide range of operating conditions.

3. *Average Activity Time.* This indicator gives an idea of the CPU time each task uses between possibly blocking primitive calls. If many tasks have rather lengthy activity times, it may indicate the need for time sliced scheduling or for preemptive scheduling.

4. *Number of Context Switches* and *Number of Task Preemptions.* These measures give an idea as to the cost associated with time sliced or preemptive scheduling.
The use of this indicator requires measuring the number of context switches without any type of preemption and then determining the increase when preemptive or time sliced scheduling is used. The number of task preemptions gives an indication as to the number of context switches attributable to preemption or time slicing.

5. **Task - Response Mapping.** This indicator identifies how many as well as which responses are associated with a particular task. This information is instrumental in assigning scheduling priorities to tasks. Tasks that are part of many different responses or part of urgent responses should warrant higher priorities.

6. **Number of Ready Tasks.** This indicator provides feedback on the effectiveness of the current task priority assignments. If priorities have been properly assigned, the average number of ready to run tasks should be quite small. As a rule, tasks that are consistently ready to run should be considered high frequency tasks and perhaps be assigned higher priorities to prevent bottlenecks.

7. **Message Queue Attributes.** This statistic identifies tasks which habitually have large numbers of incoming messages queued. To prevent these tasks from becoming bottlenecks, it may be necessary to increase their priorities.

8. **Internode Messaging Data and Shared Memory Access Conflicts and CPU Idle Time.** These indicators serve to determine the effects of localized and global scheduling. If global scheduling is implemented, the increase in internode messaging or shared memory conflicts can be compared to the benefits gained in terms of load balancing across processors.

9. **Response Completion Data.** This set of indicators is the ultimate yardstick by which the success or failure of specific scheduling decisions must be measured. This set of statistics must be obtained under varying event arrival rates to determine the applicability of a particular scheduling scheme to a particular set of environmental conditions.
The decisions that need to be made during the design phase of the development cycle have been described along with the statistical indicators needed to substantiate them. The next chapter describes how these indicators can be gathered in the most realistic manner possible. That is, as close to actual running conditions as possible through the use of a prototyping tool.
Chapter 5

The Multitasking Design Assist Simulator

In order to assess the effects of the various design phase decisions through the analysis of indicators, it is necessary to model the proposed system and collect statistics on its behavior. This chapter introduces a design prototyping tool and describes how it can be used to perform this modelling and statistical analysis.

A prototyping tool that is to be useful for assisting in the design of a real-time system must meet a wide variety of requirements and must take numerous factors into consideration. The chapter begins with a discussion on these requirements and focuses on the motivations for the development of the Multitasking Design Assist (MDA) simulator. This is followed by an overview of each of the components of the simulator. It should be noted that this chapter includes only a high level description of the MDA tool including its major characteristics, and the reasons behind its manner of implementation. A detailed user guide for the tool is included as appendix A.
5.1 Motivation and Requirements

Over the years, many different tools and methods have been proposed to assist in the development of real-time systems. Many of these tools are useful in defining the functional requirements from the requirements specification and others are useful in verifying that a proposed system will actually meet these functional specifications [Mur89, Age79, Rae85]. Others such as Buhr’s machine charts [Buh84] are useful in defining the structure that a particular design will take. However, there is a general lack of tools that can assist in the definition of the design phase details.

Attempts have been made at such tools through the use of discrete event simulations which involve the building of an abstract model of the proposed system. In this type of simulation, only those features and properties essential to the system’s functions are abstracted while the rest are ignored to keep the model to a manageable size [GGZ86]. For discrete event simulations, systems are modeled using a sequence of countable events, assuming that nothing of interest takes place between events. While this approach may have its merits in providing functional verification of a system, it is of extremely limited use for modelling the effects of design level details. This is especially true in light of the fact that it is precisely these details that are left out of a discrete event simulation. What is required for making design phase decisions is a tool that is capable of abstracting both the event occurrences as well as the actions that they trigger within the system.

Because the application, operating system, and hardware components are so closely interrelated in real-time systems, the main purpose of a design level simulation tool is to provide an environment in which the functions of each of these components can be monitored in an integrated fashion. To meet this end, the tool must furnish a prototyping language, development system, and run-time environment that is capable of modelling variations of each system component. This in turn, requires meeting the following objectives:

1. Allow stepwise refinement to facilitate modelling at different levels of abstraction.
2. Provide a modelling language that uses familiar programming constructs and is flexible enough to describe any system.

3. Provide a method that allows the system designer to customize various aspects of the model to reflect application specific features.

4. Provide the designer with facilities to manipulate the prototype while it is running so that the effects of anomalous situations can be observed.

5. Collect applicable statistics on the effects of various design decisions implemented in the prototype.

The first of these requirements stems from the fact that at different stages of the design phase, different levels of detail will be known about the system. At the start of the design phase, there is only a functional specification which is void of any lower level details. In order to be able to later define these details, the language must allow the initial high level design to be modeled. As the details become more evident from the decisions made through observations of the higher level prototype, a new more detailed prototype must be constructed. These iterations are continued until the final prototype is very close to the level of the actual implementation.

In order to be effective in such a process, the prototyping tool must be capable of constructing both coarse and finely detailed models as well as all the intermediate gradations. Ideally, the tool should be such that at the finest level of refinement, its representation of the system resembles closely that of existing implementation languages so that relevant parts of the prototype can be easily ported to the final system.

The second objective defines the attributes required in the design tool’s prototyping language. In order to be useful, this language should embrace some of the features common to existing languages with which the designer might be familiar. These include high level control constructs, a variety of data types, and the provisions for user defined data structures.

Because the language is going to be used to model a large variety of different systems, each with their own processors, memory, and I/O devices, the language
must be flexible and not make any assumptions regarding hardware configurations. The easiest way to do this is to let the designer construct his or her own application specific interrupt handlers and communication primitives using the basic building blocks of the language. The prototyping language itself should provide low level instructions as well as some high level facilities such as queue management routines that are often used in real-time programming.

The design of models using a prototyping language should be no less structured than that of actual systems using "real" languages. Structured designs of application software often use a state machine model to represent the internal workings of individual tasks. As such, this model should also be the basis of the prototyping language. If well defined structure is enforced during the early stages of design, it will inevitably carry through to the final implementation. Also, if the design prototypes have structures that are similar to the intended final implementation, the behavioral statistics gathered from the prototypes will be much closer and much more applicable to the actual implementation.

The third objective goes beyond the prototyping language itself and refers to the ability of the designer to be able to model various multiple processor configurations as well as various kernel functions and primitives by changing elements of the runtime environment. These elements include the ability to limit shared memory access to specific processors, the ability to associate specific event interrupts with specific processors, and the ability to model the environment.

This last factor is particularly important in light of the close interactions that a real-time controller has with its environment. A prototype that is constructed in isolation from its intended environment is useless in predicting system behavior in the real world. This is because the statistics gathered from such a prototype will not have been influenced by conditions that will be encountered in the actual deployment.

The fourth objective also in part deals with the ability of the designer to model environmental conditions. It is not enough to model the simple expected environmental occurrences to get a complete picture of how the proposed system behaves. In many instances, it is necessary to change conditions while the system is running in order
to observe its adaptability, to observe its reactions to unexpected spurious events, to observe its reactions to partial or complete errors and failures, to analyze the causes of incorrect behavior, and to confirm that correct observed system behavior is due to proper internal functioning rather than to fortunate circumstances.

To simulate the spontaneous occurrence of errors and to analyze the internal operations of a system requires the ability to temporarily stop the model, examine and/or modify some of its internal characteristics or some environmental characteristics, and then continue the simulation from the point of interruption. To meet this requirement, the prototype tool must come with a reasonable man machine interface as well as a good development support system.

The fifth design tool objective is perhaps the most important as the other objectives serve only to allow the designer to easily acquire a complete statistical picture of the proposed system. The ultimate reason for designing through prototyping is to gain behavioral information prior to commitment to a particular design.

5.1.1 System Limitations

The Multitasking Design Assit tool that will be proposed in this chapter meets all of the above objectives but it also has its limitations. As with all simulations, it is impossible to acquire absolute performance statistics on any aspect of the design. This is because such measures have extremely high degrees of variance depending on the implementation hardware characteristics, on the chosen implementation language, and on the way the actual code is written. In addition, since the MDA simulator uses a single processor to model multiple processor systems and because all hardware functions are simulated in software, it is impossible to get accurate time based statistics.

However, in realization of these limitations, the statistical focus of the MDA simulator is not on time based indicators but rather on counters. Because it is impossible to provide absolute timing statistics regarding items such as communications and context switches, the MDA tool simply counts the number of times various system events occur. The accuracies of the gathered statistics tend to improve greatly as
more details are introduced into the prototype. If the actual timing statistics for a particular hardware setup are available from manufacturer's specifications, then the counter values can simply be extrapolated using the hardware specific data.

Another current limitation of the MDA tool is that its main purpose is specifically to model systems at the design phase of the development cycle. While it can also be used to analyze a functional specification, it is certainly not the best tool for this purpose. It would be of great advantage if the same tool can be used for all phases of the development cycle: from the functional specification through to the final implementation. This uniformity not only saves the designer from having to learn several different tools, but it also reduces the number and degree of discrepancies that can arise in transferring a system from one toolset to another at each development phase.

Because the MDA tool has been designed as a type of skeleton tool that can be customized for any application, it could very well be extended or modified to act as an executable functional specification tool. These modifications may include a preprocessor that facilitates translation of functional specifications to the MDA language. This thesis however focuses only on the feasibility of a design phase tool and leaves the possibility of extensions for functional specifications to further research.

The MDA simulator consists of three related components: a prototyping language, a development support system, and a run-time environment. These components are discussed in the following sections starting with the run-time environment and the general MDA architecture.

5.2 The MDA Architecture

The MDA tool provides a system composed of several virtual machines on which the application and kernel software can run. All software is specified in a fairly high level assembly language (the MDA language) which is interpreted directly by the virtual processors\(^1\). Each of these virtual machines is based on the uniprocessor model

---

\(^1\)In this discussion, the term "virtual processor" is used to describe the processors defined in the system prototype while the term "actual machine" refers to the real machine running the simulation.
provided by the KASM (Kernel Assembler) system [Kar88]. All parts of the MDA system have been implemented in UNIX/C and also in Turbo C. As such the tool can be run both on DOS based machines as well as UNIX based machines. There are three main aspects to the MDA architecture: the memory and address layout within processors, the interlink scheme between processors, and the execution engine of the system.

5.2.1 The Memory Structure

An MDA simulation is actually a two step process. The first step involves compilation of the MDA language into an internal code structure that can be interpreted by the MDA virtual machine. The second step is the actual interpretation process. To allow fast simulation times, all of the compiled code for each virtual processor is resident in the main memory of the actual machine at all times during the simulation. Thus the number of tasks and virtual processors is limited only by the memory capacity of the actual machine.

Each virtual processor is logically divided into five separate memory areas:

1. Application code
2. Application data
3. Kernel code
4. Kernel data
5. Data global to processor

The application code consists of the instructions of all the application tasks resident in the processor. The code for each task is loaded starting at address 0 and continues until the end of the task code is reached or until address 1000 is reached. That is, each task starts at logical address 0 and can have a maximum of 1000 instructions. The maximum number of tasks per processor is a user modifiable constant.

Along with its memory requirements for code space, each task also has private data associated with it. This data is accessible only to the task in which it is defined.
As each task is loaded into the virtual machine, space is allocated for each of the variables defined within the task. Each task is allotted its own local data space. The MDA system does not provide any facilities for dynamic memory management. As such, all data must be defined in a static fashion prior to use. During execution, a context switch automatically changes the task data set that is accessible to the newly executing task. Context switches are performed by the CXTSW\textsuperscript{2} instruction of the MDA language.

Like the application software, the kernel software must be provided by the designer. The kernel code consists of the instructions used to define the operating system of the virtual processor. However, unlike the application code, there is only one kernel per processor. The kernel code is contained within a single task which is loaded starting at logical address 1000 and can contain up to 1000 instructions. All data associated only with the kernel is treated as local data for the kernel task. This data can only be accessed by the code included as part of the kernel.

The final memory area is allocated to data that is global to the processor. All tasks as well as the kernel can access this data at any time without the need for any type of context switching. Data is considered to be global if it is defined outside the syntactic scope of any task. Any type of data, be it global or local is defined in terms of named variables and constants. If the same name is used for a data item local to a task and for a global data item, the local item is accessed when the common name is encountered within that task.

To facilitate the transfer of information from the run-time environment to the user written application and kernel software, six special global system variables are predefined in each of the virtual processors and initialized by the MDA loader. The functions of these variables is outlined in the following list.

1. \textit{TASKSTK} - This stack variable represents the stack associated with each task in the processor. There is one stack for each task. When a context switch occurs, the stack of the old task is saved and the stack of the new task is assigned to the \textit{TASKSTK} variable.

\textsuperscript{2}Details of all MDA instructions can be found in appendix A.
2. **CURRTASK** - This is a string variable that contains the name of the task that is currently executing.

3. **TASKSTRT** - This variable contains the start address of the currently executing task. Because the MDA language requires that all subroutines be defined before use, the start address of the mainline of a task is not known until the task has been loaded into memory.

4. **CURRPRIORO** - This integer variable provides the current priority of the presently executing task.

5. **TASKLIST** - This queue variable contains the name of every task resident in the processor. The names are queued in the order that the tasks were loaded.

6. **PC** - This integer variable is the program counter associated with the currently executing task. It contains the address of the next instruction to be executed in the task. When a context switch occurs, the PC of the old task is saved and the program counter of the new task is assigned to the PC variable.

The memory configuration of each virtual processor is illustrated in figure 5.1.

### 5.2.2 Virtual Processor Interlink Structure

The interlink scheme between the individual virtual processors is quite straightforward in that the underlying structure is simply a single shared memory. Data items can be placed in this shared memory by one processor and accessed by the other processors. The only restriction is that all data items to be placed in the shared memory must be defined before they are used. This allows the MDA tool to allocate a fixed area of shared memory for each shared variable prior to the start of execution. The shared memory model was chosen as the underlying interprocessor communications medium because it allows user simulation of both shared memory and message passing type communication schemes.

The access rules for shared memory are similar to those for global data except that any task on any processor in the system can access the shared memory at any
time. For any variable access, the task’s local data is searched first, followed by global processor data, followed finally by the shared data. As a result, a shared structure can be accessed only if no structure of the same name exists in the accessing task’s local or global data space. Because multiple processor execution is interleaved on a single machine in the MDA environment, it is possible for several processors to access the same shared data item simultaneously. To prevent collisions of this type, the MDA language provides a LOCK instruction which operates much like the “test and set” type instruction found on real processors. This instruction essentially locks out other processors from shared memory access until the current access is complete (as signalled by the UNLOCK instruction). The interlink scheme between virtual processors is illustrated in figure 5.2.

5.2.3 Virtual Machine Execution

The execution engine of the MDA simulator is responsible for several things including the actual interpretation of the loaded instructions, determination of which events are
to be raised at what times, and the interleaving of a single real processor between a
number of virtual processors. The pseudo code for the execution engine is given in
figure 5.3.

After all of the application and kernel code is loaded into each virtual processor,
the simulation can be started. The MDA environment (as used in this thesis) allows
from 1 to 5 processors to be specified in the system. (The maximum number of
allowed processors can be configured by the user and would depend on the amount
of memory available in the actual machine). If the system consists of more than one
processor, then the execution engine simply cycles through all processors executing
one instruction per processor per cycle. Each instruction uses one time unit on its
virtual processor. There is no overhead associated with changing virtual processors
after every instruction as the code for all processors is always resident in memory.
The advantage of switching processors after every instruction is that a more realistic
sequencing of the multiple processor execution stream is achieved.

The only circumstances under which multiple instructions are consecutively exe-
cuted for a single virtual processor are when there is only one processor in the sys-
tem, or when the LOCK instruction is encountered. If a processor executes a LOCK
instruction, that processor is allowed to continue execution uninterrupted until an
UNLOCK instruction is encountered.
Initialize System Data
Compile and Load All User Defined Software

Set current virtual processor
WHILE not end of simulation time DO
  Get environment of current virtual processor
  Check for interrupt on current virtual processor
  IF interrupt occurred THEN
    Push PC onto TASKSTK
    PC <- Address of interrupt handler
  ENDIF

  Fetch instruction located at PC
  PC <- PC + 1
  Execute instruction

  Update shared memory access statistics
  IF not running locked AND
    not executing statistical instruction THEN
    Increment current virtual processor
  ENDIF
ENDWHILE

Figure 5.3: Execution Engine Pseudo Code

The last major function of the execution engine is to check for processor interrupts. MDA allows up to six separate interrupts to be associated with each virtual processor. Each time through the instruction execution cycle, each of the six interrupts associated with the current processor is checked to see if it is active. The times at which individual interrupts are asserted are determined by the characteristics of the events associated with the interrupts. If an interrupt has been raised, the current program counter is saved on the TASKSTK and the PC is set to the address of the appropriate user defined interrupt handler code. These interrupt handlers must be coded as part of the processor kernel.
5.2.4 Kernel Structure

A processor kernel defined in the MDA system consists of a number of elements. The entire kernel is syntactically contained within a single task and consists of local kernel data, definition of shared data items, definition of subroutines, definition of system primitives, interrupt handlers, and the task mainline code. The structure of a typical kernel is given in figure 5.4.

The local kernel data consists of items such as the control blocks of all the application tasks, possibly IPC buffers, possibly semaphore structures, etc. These items are accessible only to the kernel and are used for system related functions. All of this data is defined by the user. The other class of data definition within the kernel is the definition of shared data. This is the data that is accessible to any task on any processor in the system regardless of the kernel in which it is defined. Shared data variables can be defined in the kernel of any processor. If two kernels define a shared data item with the same name, a warning is issued and only the definition contained in the kernel loaded first remains valid.

The MDA language is procedure based and as such, procedures can be defined in the kernel task. All procedures must be defined before they are used. Any procedures defined within the kernel task can be called only by the kernel task mainline, by other procedures defined within the kernel task, and by primitives defined within the kernel task. Primitives are like procedures with two notable differences. The first is that primitives are well known throughout the processor and can be called from any application task resident on the same processor. The second is that primitives can be defined only in the kernel task.

Kernel primitives can be defined for two reasons. The first is that they are to act as functions that can be called by application software to perform operations such as IPC. The second reason is that they are to act as interrupt handlers. Only primitives can be used as interrupt handlers. Once an interrupt is raised, control is transferred to its associated primitive. Thus all interrupts must first be channelled through the kernel. In this way, the kernel has control over things such as which task(s) should be readied as a result of the interrupt, what data should be saved, and whether task
KERNEL <kernel name>;
TASK <kernel task name>, <priority>;

TYPE <kernel user defined type declaration>;

TYPE <kernel user defined type declaration>;

VAR <kernel variable declaration>;

VAR <kernel variable declaration>;

SHARED <shared variable declarations>;

SHARED <shared variable declarations>;

PROC <procedure name>;

.Local Procedure Code

ENDPROC;

PRIM <primitive name>;

.System primitive code

ENDPRIM;

PRIM <primitive name>;

.System primitive code

ENDPRIM;

PRIM <event handler name>;

.Event handler code

ENDPRIM;

BEGIN {Kernel Task Mainline}

.System Initialization code

ENDTASK;
ENDKERNEL;

Figure 5.4: Kernel Structure
preemption is warranted.

Finally, the mainline of the kernel task is used primarily to initialize system data as it is the first user coded software that runs on processor startup. Kernel functions such as scheduling can be driven from an interrupt or they can be part of the kernel task itself. Since the main goal of the MDA tool is flexibility, the details of all kernel applications and functions are left entirely to the system designer. The designer can choose to model any type of operating system primitive that he or she can devise.

In addition, MDA allows different user defined kernels to run on different virtual processors during a simulation. In this way, the interactions of different types of kernel functions can be directly observed. This flexibility in kernel design does not mean that a system designer always has to program a kernel if it is only the application software behavior that needs to be analyzed. It is possible to define standard primitives such as the semaphore signal and wait or the standard Harmony primitives send, receive, and reply and keep them in a permanent kernel file. This standard kernel, once defined, can then be used as though it is a built-in part of the MDA tool as far as the application software is concerned.

5.2.5 Application Structure

The application software is divided into a number of tasks that can execute concurrently with other tasks. In the MDA system, the form of these application tasks differs greatly from that of the kernel task. The first difference is that unlike the kernel task, application tasks can not have primitives defined within them nor can they define shared data. The second difference is more profound and concerns the actual structure of the application.

Because of the complex and diverse nature of the operations performed by a real-time kernel and because the main kernel task is usually only used for initialization, MDA allows a great deal of freedom in how it can be coded. However, this is not the case for the application tasks. In order to promote a well structured design right from the prototype stage, MDA requires all application tasks to conform to a state machine representation. A typical state machine representation of a task is given in
In this figure, states A, B, and C represent the places at which a task may be suspended due to the actions of system calls. Hence these states represent calls to potentially blocking IPC or synchronization primitives, device handler calls, or any action that may temporarily impede the progress of the task. The transitions between the states represent the individual activities that make up the task. These activities perform the basic functions of the task and once started, they do not require any external data, services, or resources and are therefore atomic entities. In many instances, it is possible to have several transition arcs emanating from a single state. To allow this situation, conditions are placed on the execution of each of these arcs.

This state machine representation imposes an implicit structure on the design of the application tasks. This structure is shown in figure 5.6.

As this is a direct translation of any task's state machine representation, it is also the basic control structure chosen for the MDA language. The only modification to the basic structure is that the inner case construct has been replaced by a series of if..endif constructs to allow different conditions to be tested within each state. An example of an application task coded in the MDA language is given in figure 5.7. In order to fully comprehend this type of example, it is necessary to introduce some
TASK
   LOOP
   CASE current state OF
   State A: CASE condition OF
      1: Do activity 1
         Call system primitive
         Set next state to B
      2: Do activity 5
         Call system primitive
         Set next state to C
   Endstate

   State B: CASE condition OF
      1: Do activity 2
         Call system primitive
         Set next state to B
      2: Do activity 3
         Call system primitive
         Set next state to C
   Endstate

   State C: CASE condition OF
      1: Do activity 4
         Call system primitive
         Set next state to C
      2: Do activity 6
         Call system primitive
         Set next state to A
   Endstate
   ENDCASE
   ENDLINK
   ENDTASK

Figure 5.6: Application Task Structure
details of the MDA language.

5.3 The MDA Language

This section provides an overview of the MDA language in just sufficient detail to allow the reader to comprehend the examples presented in subsequent sections. A detailed user guide is included as appendix A. The discussion on the working of the MDA run time support facilities is also left to the user guide.

The MDA language is similar to many mnemonic based assembler languages with three significant exceptions. The first is the inclusion of high level control constructs similar to those found in languages such as C or Pascal. The second is the inclusion of powerful "meta" instructions that allow high level operations on abstract data types or complex system operations such as context switches. The final deviation from standard assembler languages is that the MDA language directly supports the definition and access of abstract data types.

5.3.1 Data Types

There are five base data types in the MDA language: integer, boolean, string, stack, and queue. Variables can be defined as being any of these types while constants must be one of integer, string, or boolean. Variables must be defined before they are used by means of the VAR directive as illustrated in the examples of figure 5.8. Constants on the other hand can not be explicitly declared but are coded as instruction operands. A constant value is always preceded by the '#' character. Thus the operand #764 represents the constant integer -764.

The size of the integer type is dependent on the size of integers on the machine hosting the simulation. Typically, the valid range of integers will be not less than -32767 to +32768. Only assignment, arithmetic, arithmetic comparison, and I/O type operations are allowed on integer data types.

The internal representation of the boolean type is based on the integer values 0 (false) and 1 (true). As such, a boolean type requires the same amount of space as
VAR msg1, t_msg_alt;
VAR msg2, t_msg_alt;
VAR flag, integer;

ACTIVITY get_rand;
    rand #1, flag;
ENDACT;
ACTIVITY a1;
    ticks #200;
ENDACT;
ACTIVITY a2;
    ticks #40;
ENDACT;

BEGIN
    DO_ACT get_rand;
    CALLP nilprim;
    SETSTATE #/start/;
    LOOP;
        STATESELECT;
        STATE start;
            IF #true;
                DO_ACT get_rand;
                CALLP waits, #0;
                SETSTATE #/acts/;
            ENDIF;
        ENDSTATE;
        STATE acts;
            IF (flag = #0);
                CALLP nilprim;
                SETSTATE #/act1/;
            ENDIF;
            IF (#true);
                CALLP nilprim;
                SETSTATE #/start/;
            ENDIF;
        ENDSTATE;
        STATE act1;
            IF #true;
                DO_ACT a1;
                CALLP send, #/q1/, #/msg2/;
                SETSTATE #/start/;
            ENDIF;
        ENDSTATE;
    ENDLOOP;
ENDSELECT;
ENDTASK;

Figure 5.7: Example MDA Application Task

124
VAR int_val, integer;
VAR stk_val, stack;
VAR bool_val, boolean;
VAR q_value, queue;
VAR str, string;

Figure 5.8: Example Variable Declarations

an integer. However, the MDA language does not allow the direct integer assignment of the values 0 or 1 to boolean variables. The only values that can be assigned to boolean variables are the constants #true and #false. The only operations allowed on boolean types are assignment, logical comparison, and I/O.

The string data type is the same as in any other language except that the characters in a string constant are delimited by the '/' character. The operand #/string constant/ is an example of a valid string. Strings are limited to a maximum of 39 characters. The operation types valid for strings are assignment, comparison, and I/O. As will be discussed shortly, string types are instrumental in indirect addressing.

The stack and queue types do not have any constant representations and exist as variables only. The stack is simply a LIFO list, each element of which is capable of storing any data type. That is, a single stack may contain elements of different types. The elements of a stack are dynamically allocated so there is no preset upper bound on the size of the stack. The only limitation is the amount of memory available on the actual machine. The queue data type is simply the FIFO equivalent of the stack. Only stack type operations such as push, pop, and sizes are allowed on the stack data type and only queue type operations such as addq, delq, and sizesq are allowed on the queue data type. The queue data type is used in place of conventional arrays.

In addition to the five base built-in data types, the MDA language allows the user to define his or her own abstract data types through the use of the TYPE directive. These complex data types use the five base types to build record aggregates to better organize related data. An example of such an aggregate is the task control block maintained by the kernel. The type definition of such a control block along with its
**Figure 5.9: Task Control Block Type**

The individual fields within the aggregate record are accessed as in most high level languages by separating the qualifiers with periods as in:

```
curr_tcb.run_data.rdy_time
```

This allows access to the "rdy_time" field of the task control block. A user defined data type can be nested to any level and can use any of the five base types or other user defined types in its declaration. The only operation that can be performed on a complete aggregate is the assignment operation. For one complex variable to be assigned to another, they must both be of the same user defined type. The operations that can be performed on the individual fields making up the user defined type are limited only by the type of the field itself.
<table>
<thead>
<tr>
<th>Assignment Class</th>
<th>Arithmetic Class</th>
<th>Stack Class</th>
<th>Queue Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE</td>
<td>ADD</td>
<td>PUSH</td>
<td>ADDQ</td>
</tr>
<tr>
<td></td>
<td>SUB</td>
<td>POP</td>
<td>DELQ</td>
</tr>
<tr>
<td></td>
<td>MULT</td>
<td>SIZES</td>
<td>SIZEQ</td>
</tr>
<tr>
<td></td>
<td>DIV</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAND</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System Class</th>
<th>Control Class</th>
<th>I/O Class</th>
<th>Logical</th>
</tr>
</thead>
<tbody>
<tr>
<td>CXTSW</td>
<td>CALL</td>
<td>PRINT</td>
<td>&amp;</td>
</tr>
<tr>
<td>LOCK</td>
<td>RET</td>
<td>PRINTCR</td>
<td>()</td>
</tr>
<tr>
<td>UNLOCK</td>
<td>IF</td>
<td>READ</td>
<td></td>
</tr>
<tr>
<td>CAUSE</td>
<td>ELSE</td>
<td>FPRINT</td>
<td>&gt;</td>
</tr>
<tr>
<td>ACKINT</td>
<td>ENDIF</td>
<td>FPRINTCR</td>
<td>&lt;</td>
</tr>
<tr>
<td>HALT</td>
<td>WHILE</td>
<td>FTYPE</td>
<td>&gt;=</td>
</tr>
<tr>
<td>DISABLE</td>
<td>ENDFWHILE</td>
<td>FTYPECR</td>
<td>&lt;=</td>
</tr>
<tr>
<td>ENABLE</td>
<td>ENDPROC</td>
<td></td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>BIND</td>
<td>ENDACT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TICKS</td>
<td>DOACT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRKPT</td>
<td>SETSTATE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td>STATESELECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ETIME</td>
<td>LOOP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ENDPRIIM</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ENDTASK</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CALLP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.10: MDA Instruction Set

5.3.2 Instruction Set

There are seven different classes of instructions in the MDA language as well as a number of directives. The complete instruction set broken down by class is given in figure 5.10.

The seven classes of instructions are: assignment, arithmetic, stack, queue, system, control, and I/O. The general instruction format is:

```
opcode       operand1, operand2, operand3, operand4;
```

Only a few instructions of the control class do not follow this form. The number of operands varies from zero to four. In addition, the operands can use any one of three addressing modes: immediate, direct, or indirect. The immediate mode simply refers to constant operands which can be of type integer, boolean, or string. The
IMMEDIATE : #744
   */This is a string/

DIRECT : curr_tcb.run_data.run_time
         tr
         tcb_name

INDIRECT : @tcb_name
           @tcb_name.priority

Figure 5.11: MDA Addressing Modes

direct mode refers to variables of any type defined through the VAR directive. Direct operands have no special character prefix and they may be qualified if they are of a user defined type. Indirect operands are variables that serve as pointers to other variables. However, rather than containing the storage address of the dereferenced variable, indirect operands contain the symbolic name of the variable to which they are pointing. As a result, all indirect operands must be of type string. Indirect operands are prefixed by the '@' character. An example of each type of addressing mode is given in figure 5.11.

The move instruction is the only instruction included in the assignment class. The only notable feature of this instruction is that the source and destination operands must be of the same type. Complete aggregates of user defined types can be assigned to each other as long as they are of the same type.

All of the instructions in the arithmetic class are fairly self explanatory as are those of the I/O class. In the I/O class, the fprint and ftype instructions copy data to a file rather than to a terminal. For the read instruction, the input data must be of the same type specified by the read operand.

The stack class includes the push, pop, and sizes instructions. The push instruction will push an operand of any type on to the specified stack whereas the pop instruction removes the top stack element and assigns it to the specified variable. Because a single stack can contain elements of different types, the programmer must ensure that the

---

\(^3\)Details of all instructions can be found in appendix A.
type of the element being popped is the same as the target variable. The sizes
instruction returns the number of elements contained in the stack. It is good practice
to ensure that the stack size is greater than zero before popping. The instructions in
the queue class are completely analogous to the stack class.

The system class of instructions is quite varied and includes the cxtsw, lock, and
unlock instructions whose functionalities have already been discussed. The cause,
enable, disable, bind, etime, and ackint instructions are all related to the interrupt
system. The cause instruction allows any task on any processor to raise a specified
interrupt on any processor. This capability allows “environment” tasks to be modelled
that can raise environmental events at appropriate times. The enable and disable
instructions allow the enabling and disabling of specified interrupts on the processor
on which they execute. The bind instruction allows any task, including the kernel, to
associate a primitive handler with a specific interrupt. The etime instruction records
the time at which an interrupt occurred and finally, the ackint instruction simply
clears a pending interrupt.

The halt, brkpt, time and ticks instructions are responsible for timing and control
of execution. The halt instruction simply stops the processor and terminates the
simulation. The brkpt serves as a breakpoint. When it is encountered, execution
ceases and control is returned to the MDA development support system which allows
single stepping, variable examination, etc. The time instruction allows time stamping
by returning the current virtual CPU time. The ticks instruction simply increments
the simulation time and the appropriate virtual CPU time by a specified amount.
This instruction can be used to simulate the actions of some activity if the details of
the activity are not important.

The control class of instructions defines the flow of control within an MDA pro-
gram. There are two subclasses of such instructions. The first subclass includes the
if, else, endif, while, endwhile, loop, endloop, setstate, and stateselect instructions.
The setstate and stateselect instructions are found only in the main loop structure of
an application task. The stateselect instruction acts as a case construct for selecting
an execution path based on the current task state. The setstate instruction sets the
task state for the next cycle. The "if" and "while" constructs are exactly the same as those found in high level languages. They alter the flow of execution based on the result of some logical expression. Two examples of the use of these instructions are:

\[
\text{if } ((tcb\text{-}prio >= currprio) \&\&(size = \#27));
\]

\[
\text{while (running) \&\& (~error);}
\]

The results of any logical expression must be able to take on the values \#true or \#false. In the second example above, the variables "running" and "error" must be declared to be of type boolean. Further, arithmetic comparisons can be done only on integer or string data types. The arithmetic comparison operators are \(>, \geq, <, \leq, =, <>\). The logical operators are \&, \|, -. The parts of the logical expression can be delineated with any valid bracket combination.

Since the MDA language is a procedural language, the second subclass of control instructions is concerned with the delineation of code blocks and transfer of control between these code blocks. The proc, endproc, prim, endprim, activity, endact, task, and endtask instructions serve to mark procedures, primitives, activities, and tasks, respectively. The doact, call, and callp instructions are used to transfer control to the specified activity, procedure, or primitive, respectively. Only the call and callp instructions can have parameters associated with them. In these instructions, the first operand is the name of the procedure or primitive and the subsequent operands are the parameter list. When any of these instructions is encountered, the current PC is pushed onto the TASKSTK and PC is set to the address of the named subroutine. If there are parameters (allowed only for call and callp), these are pushed onto the TASKSTK in order of appearance after the PC. It is up to the called subroutine to pop these parameters from the stack before returning. When the end of the subroutine is reached or when the ret instruction is encountered, the PC is popped from the TASKSTK and control returns to the calling code.
5.3.3 Event Types

In order to allow a complete and realistic modelling of environmental events, the MDA system allows for the specification of a large variety of event types. The events associated with each virtual processor are specified by means of an event file which contains a number of directives. There are six different directives and all must be specified for each event. The directives are EVENT, INTERRUPT, TYPE, PERIOD, PROBABILITY, and QUEUE. The EVENT directive simply serves to give a symbolic name to the event and the INTERRUPT directive associates a particular interrupt line with the event.

The TYPE directive specifies whether the event is to be cyclic, sporadic, dependent, or delayed. A cyclic event occurs at regular intervals. For this type of event, the PERIOD directive specifies the number of time units that must elapse between assertions of the event. For cyclic events, the PROBABILITY must always be 100. A sporadic event is the opposite of a cyclic event in that it has no fixed period but it occurs with a certain probability. As such, the PERIOD directive must be 0 while the PROBABILITY directive must be in the range 1 - 100. The determination of whether an event with probability less than 100 will occur during any cycle is based on a random number generator. A random number between 0 and 100 is generated. If this number is less than the specified probability, then the event will be raised during the cycle.

A dependent event is one that can be raised only through the execution of a cause instruction. A dependent event, when used in conjunction with a task simulating the environment, provides an effective means of modelling environment state dependent occurrences. A dependent event associated with any processor can be raised by a cause instruction on any processor. Thus it is possible to dedicate a virtual processor to modelling the environment while the other virtual processors model the proposed controller. Both the PERIOD and PROBABILITY of a dependent event must be 0.

Finally, a delayed event is one which always occurs some time after another event. For this type of event, the PROBABILITY directive is used to specify the interrupt number associated with the event that the delayed event is to follow. As a result, the
probability value must be in the range 0 - 5 (corresponding to the range of interrupts) and it must not be the same as the number specified in the INTERRUPT directive. That is, a delayed event can not follow itself. A delayed event can occur either a fixed amount of time after the triggering event or it can occur a random time interval after the triggering event. In order to specify a fixed lag time, the PERIOD directive should specify a value greater than 0. If the specified PERIOD is 0, then a random lag value of between 1 and 200 time units is selected. A different random value is chosen after each triggering event occurrence.

The final directive used to specify event parameters is the QUEUE directive. This directive is used to specify the number of event occurrences that are to be queued for each event. A queue size of 0 indicates that the event is volatile in nature and if not acted upon immediately, it will be lost. A queue size of 1 indicates that the event is latched. The event occurrence will be saved until it can be acted upon. Larger queue sizes indicate the number of occurrences that are to be saved. For all events with queue sizes greater than 0, the ackint instruction must be used to clear an event occurrence from the queue. If several event occurrences are queued, then several ackint instructions must be executed to clear them.
5.3.4 MDA Files

The event file is not the only specification file required by the MDA tool. The specification of a complete system requires five different types of input files. These are known as .sys, .nod, .ml, .evt, and .tsk files corresponding to the filename suffixes that each file type must bear. In addition, the MDA compiler and loader produce source listing files known as .lis files.

The input files required by MDA follow the hierarchical structure shown in figure 5.12. Each of these files consist of various directives along with the actual instructions of the MDA language. The contents of the .evt file have been discussed in the previous section. A complete example of a .evt file can be found in appendix A. The formats of the remainder of the MDA files are discussed below.

5.3.4.1 System Definition File

The .sys file contains only the NODE directive. This file is used to specify the number and symbolic names of all virtual processors that are to be included in the system. The names of the processors also must be the names of the .nod files associated with each. For example, a file named example.sys may contain the following lines:

```
NODE cpu1
NODE cpu2
NODE dsp
```

This signifies that there are three processors with names cpu1, cpu2, and dsp corresponding to virtual processors 0, 1, and 2 respectively. It also indicates that
; NODE cpu2 FILE
;

TYPE t_sema = count : integer *
    wait_q : queue;

VAR node_name, string;
VAR mutex, t_sema;
VAR buf_avail, t_sema;
VAR item, integer;

EVENT clock;

KERNEL base;

TASK task1, 50;
TASK task2, 10;
TASK task3, 50;

ENDNODE;

Figure 5.13: Example of .nod File

the specifications for these three nodes are contained in files cpu1.nod, cpu2.nod, and
dsp.nod respectively.

5.3.4.2 Node Definition File

The .nod files are used to specify five thing associated with each processor. These
are: the kernel for the processor, the global user defined types, the global variables,
the events associated with the processor, and the tasks resident on the processor.
An example of the file cpu2.nod is given in figure 5.13. The KERNEL directive is
used to provide the name of the kernel to be loaded into the virtual processor. It
also indicates that the specifications for this kernel are contained in a file named
base.knl. The TYPE directives are used to define the global user defined types that
are to be visible to all tasks in the processor. Similarly, the VAR directives define
the global data items that are to be accessible to all tasks in the processor. The
EVENT directive specifies the name of the file that contains the event specifications.
In the example, this file would be named clock.evt. The TASK directives are used to specify the names and priorities of every application task that is to be resident in the processor. The task names specified in these directives must correspond to the names of the files that contain the individual task specifications. In the example, these files would be named task1.tsk, task2.tsk, and task3.tsk. Finally, each node file must end with the ENDNODE directive.

5.3.4.3 Kernel Definition File

The .knl file is used to specify all aspects of the kernel including the actual kernel code instructions. An example of a complete kernel file is included in appendix A. The first two statements of this file must be the KERNEL and TASK directives. They serve the same functions as in the .nod file except that they do not reference any subsequent files. The TASK directive merely gives the name and priority of the kernel task. These directives may be followed by any number of TYPE and VAR directives that specify the user defined types and data items local to the kernel. In addition, the kernel may include one or more SHARED directives. These directives are similar to VAR except that they define data items that are to be resident in shared memory and thus are accessible by any task on any processor. These initial declarations are then followed by the definitions and code of local procedures and system primitives followed by the main body of the kernel task.

5.3.4.4 Task Definition File

The last type of input file is the .tsk file. These files contain the complete specifications of the application tasks in the system. They can include TYPE and/or VAR directives to define structures and data local to the task. In addition, they contain the code for all the activities and procedures of the task as well as the task mainline. The structure of the task mainline is fixed by the MDA language. A complete example of a task file can be found in appendix A.
5.3.4.5 Output Source Listing Files

As each of these files is compiled and loaded, the MDA tool creates a listing file for each loaded file. For example consider a system consisting of two processors specified by the files buf.sys, first.nod, second.nod, clk.evt, sema1.knl, sema2.knl, task1.tsk, and task2.tsk. The hierarchical structure of this file system is given in figure 5.14. Note that the same tasks are resident on both processor 0 and on processor 1. The following listing files would be produced by MDA in this case: buf0.lis, first0.lis, clk0.lis, sema10.lis, task10.lis, task20.lis, second1.lis, clk1.lis, sema21.lis, task11.lis, and task21.lis. It should be noted that a .lis file is created for every input file. If the same file is loaded into more than one processor, then more than one .lis file is produced for it. The name of the .lis file is made up of the input file name, followed by a number indicating the processor on which it was loaded, followed by .lis.

A listing file contains a copy of the input source file. Each statement in the source code is sequentially numbered and accompanied by the address at which it was loaded in the given virtual processor. Statements that do not generate any compiled code (such as ENDF) and directives (such as VAR) do not have loading address locations associated with them. The load address locations are generated as each file is loaded.
5.3.4.6 Source File Loading

The sequence of loading all the source files into the virtual processors closely follows the hierarchical structure of the file system. An example of such a sequence is given in figure 5.15. The file loading process starts with the .sys file and then is interrupted by the loading of the first .nod file specified in the .sys file. The loading of the .nod file is then temporarily interrupted by the loading of the .knl file specified in the .nod file. Once the .knl file is loaded, the loading of the .nod file continues until it is again interrupted, this time by the loading of the .evt file. In other words, loading always starts with the .sys file. As soon as a file name of a lower level file is encountered during the loading of the current file, loading of the current file is temporarily suspended to load the lower level file. Once the lower level file and all of its “child” files have been loaded, the loading of the current file continues.

The main motivations behind the MDA file system are encapsulation of functionality and software reusability. The divisions between the five MDA file types allow the system designer to independently design application software, kernel software, define event characteristics, and configure the system in terms of the number of processors. Because each of these specifications and configurations is contained within one or more discrete files, it is very simple to experiment with different setups. For example, task to processor allocation is accomplished simply by moving the TASK directives.
between .nod files. Similarly, different kernels can be tried in different processors by simply changing the KERNEL directive in the appropriate .nod file(s). These are only two examples but the same reasoning applies to all configurable aspects of the system.

The final benefit of the file system structure is that the same individual tasks, kernels, or even complete nodes can be used in different places in the system by simply copying directives. In this manner, the same task or kernel can be resident on each processor, for example. Also, standard tasks and kernels can be saved in their appropriate .tsk and .knl files for use when needed. As an example, a complete kernel including the Harmony primitives can be coded in the MDA language and supplied with the MDA system as a standard Harmony.knl file.

5.3.5 Expandability of Design

The flexibility provided by MDA at the component level through the hierarchical file system is also evident at the lower language level in the form of an easily expandable instruction set. The language and interpreter have been designed so that additional instructions can be included simply by adding them to the instruction table and including one procedure to compile the instruction into the internal MDA format and one procedure in the interpreter to simulate the actions of the instruction.

The ability to expand the available instruction set is very useful if the proposed system is to simulate specialized processors with special instructions. In addition to the straightforward extension of the instruction set, the syntax of the MDA language is quite simple to modify and requires no changes to the actual compiler. The lexical analysis of the language is done through table driven finite state machines. To change the language syntax requires only the reordering of the state machine tables. This type of change may be desirable for example, if the rigid syntactic structure imposed on the application tasks is inconvenient for a particular application.
5.4 Collection of Statistics

One of the main objectives of the Multitasking Design Assist tool is to gather statistics from the modelled prototype so that the proper design decisions can be made. In order to meet this objective, MDA has two classes of statistic gathering methods. These are referred to as builtin statistics and user statistics. Data belonging to the builtin class is automatically collected by the MDA interpreter and is fixed for all application prototypes. In realization of the fact that it is impossible to predict all the types of data that may be needed for the complete analysis of a particular application, MDA allows users to program the collection of their own statistics using the same environment and instructions used to program the system model.

5.4.1 User Collected Statistics

Depending on the application, most of the statistics collected through an MDA simulation need to be programmed by the user. The reason for this is that many of the relevant statistics depend on kernel functions or on application task specific functions. Since it is part of the MDA philosophy to keep these two aspects in the designer's domain, their functionality can not be predefined. As such, it is impossible to provide builtin statistics for them. All of the user defined statistics used in this thesis however, have been preprogrammed (appendix B) and can be retained as part of any kernel file.

Using the MDA language, in addition to carrying out their operational functions, the kernel primitives for example, can be modified to collect pertinent data each time they are invoked. This type of data collection raises two concerns. The first is that the overhead incurred by the prototype to collect its own statistics may interfere with the values of the statistics themselves. For example, if data were required on the amount of virtual CPU time used for kernel functions, this data would be inaccurate due to the kernel overhead in gathering the data. For this reason, the MDA tool incorporates a system to allow monitoring without interference.

Every instruction that is not part of the prototype but is programmed only for user
data collection should be preceded by an asterisk (*). This signals to the interpreter that the corresponding instruction is to be executed but it is not to take any virtual CPU time. Further, to prevent problems in interprocessor instruction execution timing, all such consecutive instructions on any processor are executed contiguously without the normal interleaving through each virtual processor after each instruction. Thus each block of data gathering instructions is executed as an atomic entity that takes no time to run.

The second item that needs to be addressed with user programmed statistics gathering is how the data is to be communicated to the outside world. The data collected by the prototype is stored in the data structures declared within the model through the VAR and SHARED directives. In addition to these, there is a STAT directive. This directive is used in exactly the same manner as the VAR directive. A STAT variable can be used for any purpose but at the end of a simulation run, MDA goes through all STAT variables and copies their names and values to a user specified output statistics file. If the STAT variable is of a user defined type, the value of each field is copied to the statistics file. An example of code for a statistics gathering function along with its results is shown in figure 5.16.

5.4.1.1 Simulated Time

Many of the user defined statistics used in MDA simulations need some type of time basis. In the MDA environment, each instruction execution requires 1 unit of time regardless of the type of instruction. As such, each instruction execution increases the virtual processor time by one unit. The only exception to this situation arises when an instruction is preceded by an asterisk. The execution of these instructions does not require any virtual cpu time. In addition to instruction execution, virtual cpu time is incremented by any specified value using the the ticks instruction. This instruction is used to simulate actions that require some amount of time. To facilitate user statistics gathering, timestamps are taken using the time instruction which returns the current virtual cpu time.

In order to make good design phase decisions, a number of statistics need to be
; PROCEDURE sched_rtr_task (rtr_pcb : t_pcb)
;
; This procedure accepts a process control block and
; adds it to the ready to run queue in a prioritized
; fashion.

PROC sched_rtr_task;
  *time start_sched;
  pop taskstk, rtr_pcb;
  move #/ready/, rtr_pcb.status;

  addq rtr_pcb, rtr, rtr_pcb.priority;

  if (rtr_pcb.priority > curr_tcb.priority);
    pop taskstk, ret_pc;
    pop taskstk, ret_pc
    move ret_pc, curr_tcb.pcnt;
    move taskstk, curr_tcb.stk;

    move #/ready/, curr_tcb.status;
    addq curr_tcb, rtr, curr_tcb.priority;

  *time end_sched;
  *sub end_sched, start_sched, elapsed;
  *print #/Time Spent in Scheduler: /;
  *printf elapsed;

  call run_rtr_task;
endif;

*time end_sched;
*sub end_sched, start_sched, elapsed;
*print #/Time Spent in Scheduler: /;
*printf elapsed;

ENDPROC;

Output Produced:

Time Spent in Scheduler: 24

Figure 5.16: User Statistics Gathering
collected. The following sections discuss how the MDA tool is used to collect data on each of the statistical groups outlined in figure 4.3 (Statistical Factors). As the nature of these statistics has already been explained in the previous chapter, the following sections concentrate on the method of their collection using the MDA tool.

5.4.2 The Processor Data Group

The statistics in the processor data group are collected individually for each virtual processor. Only two of these statistics are builtin and the rest are programmed along with the prototype. This group includes the following items:

1. *Number and Identities of Tasks on Processor* - This is a builtin value that is recorded immediately after all of the virtual processors have been loaded.

2. *Number and Identities of Tasks Indirectly Associated with Processor* - This is a user programmed statistic that can identify all tasks that are in communication with the tasks on a particular processor. The Event Response data (to be described shortly) gives the identities of all the tasks associated with a particular response and it shows the communication patterns between tasks on all processors. This data can be used is used to identify all tasks that require data or services from tasks resident on this processor.

3. *Percentage CPU Usage by Kernel Scheduler* - This is a user programmed statistic that can be incorporated into the kernel scheduling functions. A CPU virtual time stamp has to be taken at the start and end points of every kernel function including primitive execution, queue management, and scheduling functions. From the scheduling function time stamps, the time required for each scheduling iteration has to be computed and summed for all iterations. The final percent CPU use for scheduling can then computed by dividing this sum by the total elapsed CPU time.
4. **Percentage CPU Usage by Kernel for Nonscheduling Functions** - This user programmed statistic must be computed by dividing the total time spent in executing kernel instructions by the total elapsed CPU time. This statistic gives total CPU usage for all kernel functions. The desired data can then be obtained by subtracting the percentage CPU use for scheduling operations. Again, the elapsed time data can be acquired from time stamps.

5. **Percentage CPU Idle Time** - This user programmed statistic must be computed by determining the total amount of time spent executing a special idle task and dividing it by the total elapsed CPU time. The idle task is run only when no application task is ready to run and there are no kernel functions to be performed. A time stamp must be taken at the start and end of each idle task execution.

6. **Total Number of Context Switches** - This built-in statistic is calculated by the MDA interpreter for each processor. It is calculated simply by incrementing a counter every time the context switch instruction (CXTSW) is executed.

7. **Number of Times Each Kernel Primitive Called** - This is a user programmed statistic that must be incorporated into each kernel primitive. Each primitive should include its own counter and increment that counter each time it is invoked.

8. **Data on Messages Incoming from Other Processors** - This is another user programmed statistic that must be incorporated into the operating system. The statistic should include four types of data: the identities of all processors sending incoming messages to this node, total number of messages received from other processors, average number of messages being received at a time, and the maximum number of messages being received at one time. Assuming that the prototype messaging system has been designed so that incoming asynchronous internode messages trigger an interrupt, the required data can be collected as follows. Each time the interrupt is raised, the source processor of the message
is recorded and an internode message counter is incremented. In order to simulate message reception, a fixed time delay is associated with the reception of each message. If during this time period, the internode message interrupt is again raised, then two messages are being received simultaneously. Thus each message reception episode can involve one or more incoming messages. Data must be kept by the interrupt handler on the maximum number of messages per episode and on the average number per episode.

5.4.3 The Task Data Group

This group of statistics serves to identify the behavioral characteristics of each application task. As such, they are collected for each application task on each processor. All of the statistics associated with the task data group are user programmed. Their suggested methods of collection are detailed in the following list.

1. **Total, Maximum, and Average Time Spent in Ready State** - This data can be gathered by the kernel every time a task changes system states. As soon as a task is placed in the ready state, a time stamp is taken. Another time stamp is taken when the task leaves this state. Thus the kernel keeps track of the number of times each task enters the ready state and how long it stays in this state. From this data, the total, maximum, and average times spent in this state can be calculated.

2. **Total, Maximum, and Average Time Spent in Running State** - The method of data collection for the running state is identical to that for the ready state.

3. **Total, Maximum, and Average Time Spent in Blocked State** - The method of data collection for the blocked state is identical to that for the ready and running states. However, the data collected for the blocked state can be broken down into three categories based on the reason for blockage. If the task is blocked because it must wait for a shared memory access, the total time for shared memory blockage is incremented. A similar situation holds for blockage because
the task is waiting for a message from another task and for blockage due to the
task waiting for the occurrence of an external event.

4. Maximum and Average Task Cycle Time - This statistic indicates how long each complete application task loop takes including the time required for blocking primitives, etc. This data can be gathered by the application task itself. The first activity in the task loop takes a time stamp before it starts and the last activity takes a time stamp after it completes. From these time stamps, the elapsed time can be calculated for each cycle and from this, the maximum and average cycle times may be calculated.

5. Maximum and Average Activity Time - This is another statistic that can be gathered by the application task. At the start of each activity, a time stamp is taken. The time elapsed between consecutive time stamps can be used to calculate the maximum and average execution times associated with the activities of each task. It should be noted that the time involved in the primitive call at the end of each activity is included as part of the activity execution time.

6. Percentage Task CPU Usage - This statistic indicates how much CPU usage is attributable to each task. It can be computed by dividing the total time the task spends in the running state by the total elapsed CPU time at the end of the simulation.

7. Number of Shared Data Accesses by Task - The collection of this statistic relies on the assumption that application tasks as well as OS functions do not directly access shared data but rather go through kernel primitives that can arbitrate access to shared memory. These can be specialized primitives or they can be the standard semaphore primitives. Every time one of these primitives is called, it can increment a counter associated with the calling task. A specialized shared data access primitive can also determine which piece of shared data is to be accessed and associate task counters with each data item.
8. Number of Task Preemptions - This information can be gathered by the kernel scheduler. Each time a task is preempted due to a time slice or due to a higher priority task, the scheduler can increment the preemption counter associated with the preempted task.

9. Number of Messages Sent to Other Tasks - This statistic describes the communication patterns of each task and can be collected by the kernel primitives assigned to do IPC send operations. Whenever such a primitive is called, it can examine the destination of the message and increment a counter associated with that destination. Each task should have one counter associated with each destination.

5.4.4 The Queue Data Group

This group of statistics consists of data gathered by both built-in and user programmed collection facilities. The built-in statistics are gathered for every QUEUE type variable in the prototype as well as for event queues. The user-programmed statistics are gathered only for message queues although, as always, this is at the discretion of the designer. The information pertinent to the queue data group is described in the following list.

1. Maximum and Average Queue Length - This is a built-in statistic which is updated whenever an ADDQ or DELQ instruction is performed on any queue. In the case of event queues, the updates occur whenever an event is generated and when an ACKINT instruction is executed. The actual data produced by MDA gives the largest size attained by the queue during the simulation, the total time of the simulation, and a weighted queue length factor which is calculated as

\[ \sum_{\text{length}=0}^{\text{maxlen}} \text{length} \times \text{time}_{\text{length}} \]

where length is the length of the queue and time is the elapsed time for which the queue had a size of length. The average queue length is then be calculated by dividing the weighted length factor by the total elapsed virtual CPU time.
2. **Maximum and Average Message Queue Elements by Source** - This is a user programmed statistic that categorizes the sources of messages. For each receive message queue, the receive IPC primitive should break down the number of messages in the queue by their source. A maximum and average value for number of queued messages can be stored and updated for each possible source. The average can be calculated using a time weighting factor like that for standard queues except that length is replaced by the number of queue elements from a particular source.

3. **Average Time Each Element Spends in Queue** - This user programmed statistic shows the average time that elapses between the time an item enters a queue and the time at which it leaves. Each queue element can be defined to contain a time field. Before an application adds an element to the queue, it must fill in the time field with the current time stamp. After the each element is removed from the queue, its entry time stamp must be examined and an elapsed time calculated. A counter must be maintained that tracks the number of elements removed from the queue. This counter is updated and used to calculate the average queued time. In the case where many different tasks access the same queue, each will have to update this queue statistic. This is a builtin statistic only for event queues and is user programmed for other queue types.

### 5.4.5 The System Data Group

The system data group consists entirely of builtin statistics. These statistics deal with information that is not local to a particular processor but gives an idea of the interactions between individual processors through shared memory. The individual statistics are as follows.

1. **Number of Processors** - This data simply indicates the number of configured virtual processors and is recorded immediately after loading is complete.

2. **Number of Shared Memory Access Conflicts** - This statistic indicates the number of simultaneous accesses to shared memory that could potentially occur in the
specified prototype. During every simulation cycle, each processor executes one
instruction. If a processor is running locked, then it is allowed to execute a
block of instructions per simulation cycle. So every processor executes one or
more instructions consecutively per simulation cycle. A shared memory access
conflict is deemed to have occurred when two or more processors access "shared
data" in the same simulation cycle.

The MDA interpreter maintains two types of access conflict statistics. The first
simply counts the number of shared memory access conflicts without regard to
which piece of data is being accessed. This statistic is applicable if all shared
data was stored in a memory common to all processors. The second statistic
breaks down the access conflicts by individual items. A conflict counter is kept
for each shared variable and is incremented only when two or more processors
access the same shared variable in a single simulation cycle. This allows the user
to effectively break the shared memory into smaller blocks which are accessed
only by specific processors. Figure 5.17 illustrates this concept.

This figure shows shared memory partitioned into two separate blocks. Only
nodes N1 and N2 access the MEMORY.1 block and only nodes N2 and N3 access
the MEMORY.2 block. The form of these accesses will be MEMORY.1.ITEM2
or MEMORY.2.ITEM1, etc. With such partitioning, the number of access
conflicts for either of MEMORY.1 or MEMORY.2 will be fewer than if all
of the data items were in a memory shared by all processors. (For purposes of
comparison, both of these statistics are gathered). Thus these statistics and the
MDA ability to partition shared data provide a powerful tool for prototyping
different interlink configurations.

3. Number of Writes to Shared Memory - This statistic is simply a counter that
is incremented every time a shared data variable is modified by any processor.
Like the access conflict statistic, data is kept both on writes to any shared
variable as well as on the number of writes for individual shared data items.
5.4.6 The Event-Response Data Group

This user defined group of statistics can be used to determine the interactions between all of the application tasks in the system. In addition, it can be used to correlate event occurrences with individual task actions and eventually with a response to the environment. This response can come in several forms including: the execution of a CAUSE instruction which generates an occurrence in an "environment" task, an update of some memory item, or the sending of a message to a task modelling the environment. The statistics included in this group are the following:

1. Identification of All Tasks Associated with Each Event
2. Identification of All Events Associated with Each Task
3. Identification of All Intertask Dependencies Via Messages or Shared Data
4. Maximum and Average Time Required for Each Event-Response Completion
5. Relative Order of Response Completions

6. Number of Completions of Each Type of Response

This type of data collection is very application specific and must take the semantics of various occurrences into account. For example, to gather such statistics, it is necessary to be able to identify a final response. This can be done only with the knowledge of the application. As such, all of these statistics are user programmed but in a more elaborate way than the other user programmed statistics. The collection of event response data statistics involves the use of message traces and statistical primitives. These are described in the next sections.

5.5 Execution Path Traces

In real-time systems, the occurrence of an environmental event causes the system controller to perform a number of activities which culminate in some sort of responses to the environment. However, in practice, this sequence may be more involved. For example, it is possible that a single event will give rise to a number of responses. Conversely, it is possible for a group of events to give rise to only a single response. Further, the activities performed by the controller in responding to the event(s) are usually implemented in several different tasks which may be resident on different nodes.

In order to get accurate and complete information on the actions performed by the controller in servicing an environmental event, it is necessary to track two things. The first is the interactions between the tasks involved in providing the response and the second is associating a specific event occurrence with the response actions of each task. This second item can become quite complex when viewed in light of the fact that multiple instances of the same event can occur in rapid succession. Although it is likely that the same tasks will handle each occurrence, with prioritized scheduling and nondeterministic IPC times there is no guarantee that they will be handled in the order of occurrence by each task.
The MDA tool takes the approach that each task interaction has a history associated with it. That is, a particular sequence of events have gone into producing each message active in the system as well as in producing the current contents of each shared data item. These are the two basic ways in which tasks interact with each other in servicing an event: via messaging and via shared memory. Figure 5.18 illustrates this concept.

In this figure, actions 3 and 5 represent the sending of messages between tasks T1, T2 and T3, T4 respectively. Action 7 is the generation of a response, 4 is the reading of shared data and 6 is the writing of shared data. In this example, every message 3 has the history:

1. Event 1 occurrence at time z1.
2. Processing by task T1 at time z2.
3. Message 3 sent at time z3.

Although there are other interaction methods such as the generation of events between tasks, their inclusion in the following discussion is implicit. The methods discussed are equally applicable to other task synchronization and communication methods.

Ordinarily the shared data would be protected inside a task but this has been omitted for simplicity.
Similarly, every message 5 has its unique history:

1. Event 2 occurrence at time \(y1\).
2. Processing by task T3 at time \(y2\).
3. Message 5 sent at time \(y3\).

If task T4 updates the data item D1 as a consequence of the message received from task T3, then the content of D1 has its own unique history that must be updated every time its value is changed. The history of D1 in this case would be:

1. Event 2 occurrence at time \(y1\).
2. Processing by task T3 at time \(y2\).
3. Message 5 sent at time \(y3\).
4. Message 5 received by T4 at time \(y5\).
5. D1 updated by T4 at time \(y5\).

It should be noted that the history of data item D1 includes the history of the message produced by T3. Thus the parent actions are propagated through the response. Finally, the complete history of the response (7) is:

1. Event 1 occurrence at time \(z1\).
2. Processing by task T1 at time \(z2\).
3. Message 3 sent at time \(z3\).
4. Message 3 received by T2 at time \(z4\).
5. Data D1 read by T2 at time \(z5\).
6. (a) Event 2 occurrence at time \(y1\).
   (b) Processing by task T3 at time \(y2\).
Figure 5.19: Data History

(c) Message 5 sent at time $y_3$.
(d) Message 5 received by T4 at time $y_5$.
(e) D1 updated by T4 at time $y_5$.

7. Response produced by task T2 at time $x_6$.

The final trace contains a record of all the actions involved in providing the response along with the times of their occurrence. The question of associating a particular action with a particular response can be resolved by this trace method. Consider the task depicted in figure 5.19. This task provides two separate functions. Function A is triggered when a message of type A is received and function B is triggered when a message of type B is received. Function A involves the sequence: send message A1, send message A2, and then write to shared data D1. Function B requires the sequence: send message B1, read shared data item D2, and then send message B3. Because tasks are sequential engines with a single thread of control, only one of these functions can be done at a time. For this example it is assumed that the function must complete before another is started and after completion of each function, the task cycles and waits for an incoming message of either type.

Because each of the task functions occurs as part of a response, the task must propagate the history of the response that it is currently working on. It does this in the following manner. As outlined above, every message has a history associated
with it. Whenever a function is started by reception of message A or B, the history associated with the message is made to be the current working history of the task. Assume the task in figure 5.19 receives message A with history X. The current working history of the task is now X. The first action it does is send a message A1. This action must be appended to the current working task history and then this current working task history must be associated with the outgoing message. Thus the history of message A1 will be:

1. X.


Similarly before message A2 is sent, the action must be appended to the current working task history before it is associated with the outgoing message. The last action is to write to shared data item D1. Just prior to this write, the current working task history is:

1. X.


3. Message A2 sent at time z2.

4. Data write to D1 at time z3.

When the write is done, this is the history that will be associated with the new value of data item D1. As more actions are performed by the task, the current working task history increases in size. However, whenever the task completes its functional cycle and starts a new one by receiving message A or B, the current working history is erased and replaced by the history of the newly received message. This data flow approach to tracing event response sequences is applicable to all types of task interactions and internal task structures. In addition, the final response trace can be used to capture all of the statistics of the event-response data group.
5.6 MDA Statistical Primitives

The MDA tool provides specific functions that are used to gather response traces in the manner described above. While these functions and their associated data structures are not a basic part of the MDA tool, they are programmed using the MDA language as functions that can be called by the modelled kernel. These functions are available as MDA library routines which only collect statistics and as such, do not take any time to execute (all of their instructions are preceded by "*"). This section discusses the details of these functions as well as the data structures involved in collecting all of the user defined statistics discussed in section 5.4.

5.6.1 Trace Data Structures

A response trace consists of information regarding a sequence of actions and the times at which they occurred. The logical ordering of these actions as well as the association of such a sequence with a triggering event is defined in the form of a list. Every element of such a list contains information regarding a single action and each complete or partial list represents a single response occurrence or part thereof, respectively. Further, it is quite natural to build such a list as more and more actions are performed in the progression of a response. A definition of the structure of such a list element along with its MDA language representation is given in figure 5.20.

The action field of this data structure shows the type of operation associated with the trace list element. This field can take on any of the following values:

1. *Message Sent* - This action represents the placing of a message into the queue of the receiving task.

2. *Message Received* - This action represents the removal of a message from an incoming message queue.

3. *Data Write* - This action represents the modification of a shared data item.

4. *Data Read* - This action represents the use of a shared data item.
t_trace_elt.action Values:  
1 - Msg Sent  
2 - Msg Accepted  
3 - Shared Data Write  
4 - Shared Data Read  
5 - Response Out  
6 - Event Trigger  
7 - Start Read Delimiter  
8 - End Read Delimiter  
9 - Start Msg Delimiter  
10 - End Msg Delimiter

TYPE t_trace_elt =  
    node_name: string *;  
    task_name: string *;  
    action: integer *;  
    time: integer *;  
    misc1: integer *;  
    misc2: integer;

Figure 5.20: Trace List Element Structure

5. **Response Out** - This action represents the completion of a response and signals the end of a message trace list. The *misc* field of the element structure in this case specifies the type of response that was produced and can take on any application specific integer value.

6. **Event Trigger** - This action is always the first element of a trace list. It represents the event occurrence that triggered the end response. The *misc* field of this element structure indicates the interrupt number associated with the event.

7. **Start Read Delimiter**

8. **End Read Delimiter** - These delimiters are simply marker elements in the trace list that delineate the history of a shared data item that was read by the task in question. Recall that each such item has a trace history associated with it that represents the actions performed to generate the data value. The delineation serves only to assist in the reading of the response trace.

---

⁶A particular event can trigger more than one response or a sequence of responses.
TYPE t_msg_elt = dest : string *;  Destination task
  source : string *;  Source task
  data : t_struct *;  Message data
  trc_q : queue;  Trace list of trace elements

TYPE t_data_elt = data : t_struct *;  Shared data structure
  trc_q : queue;  Trace list of trace elements

TYPE t_tcb = name : string *;  Task name
  status : string *;  Task state
  priority : integer *;  Priority
  pcnt : integer *;  Program counter
  stk : stack *;  Local task stack
  trc_q : queue;  Trace list of trace elements

Figure 5.21: Fields Using Trace Lists

9. Start Msg Delimiter

10. End Msg Delimiter - Like shared data items, each incoming message has a
    history associated with it. These list element types delimit these histories within
    the main response history.

Thus there are ten possible types of action elements in a trace list. It has been
shown that a trace list is a progress report of a response at any point in time. It has
also been shown that such progress reports need to be maintained as part of three
entities: individual tasks, individual shared data items, and individual messages. The
trace list of any task shows what point it has reached in the servicing of a response.
For this reason, every task control block (TCB) contains a field that holds the task's
current working history trace list. Similarly, the data structure used to specify any
message must contain a field that holds the history of the message in the form of a
trace list. Finally, the data structure associated with each piece of shared data must
have a field that can contain the trace list history of the current data value. These
three data structures are shown in figure 5.21.
5.6.2 Trace Procedures

In order to standardize the creation, maintenance, printing, and deletion of response traces, facilities have been programmed in the MDA language as procedures that can be called by user defined kernel primitives. A single procedure named *response update* that performs all of these functions is included in each MDA kernel file. The pseudo code for this procedure is given in figure 5.22.

There are several things of note in this procedure. The pseudo code has been expanded in some places to actual MDA code. This was done to provide some familiarity with the instructions and structure of the language. It should be noted that the number and types of parameters passed to this procedure vary depending on the type of action requested. The calling primitive must push the proper parameters on to the stack before the procedure call. Another thing that should be noted is that all of the instructions in this procedure are prefixed by '*s' so that they do not take any virtual processor time to execute. All such procedures used for statistics gathering are made nonintrusive in this manner.

The procedure starts by filling out the common fields of the new trace element such as the node on which action occurred, the task which performed the action, and the time of the action. From this point, the procedure can be treated as a case statement based on the type of action performed. There are six such actions and each of these cases will be discussed individually.

In the case of a message send, the new trace element is simply added to the task's current working history list. A copy of this list is then included as part of the outgoing message. In the case of message receive, the "message receive" trace element is added to the current working history of the task and then an incoming "start message" delimiter trace element is added to this list. After the start message delimiter, the trace elements of the incoming message history are appended one by one to the current working trace. Finally, an "end message delimiter" trace element is added. Thus the message history has been incorporated into the current working history and the message history list is deleted.

An analogous situation occurs with shared data write and read. In the case of
PROC response_update;
Get action parameter from stack
Get new trace element
Fill in node name and task name fields
Get time stamp and fill in time field
Fill in action field with parameter

*if (action = #1); SEND MSG
  *pop taskstk, dest_elt_name;
  *pop taskstk, msg_elt_name;
  *addq trace_elt, curr_tcb.trc_q, #1;
  ; COPY TASK HISTORY TO OUTGOING MESSAGE
  *move curr_tcb.trc_q, temp_q;
  *sizeq temp_q, q_size;
  *while (q_size > #0);
    *delq temp_q, trace_elt;
    *addq trace_elt, @msg_elt_name.trc_q, #1;
    *sizeq temp_q, q_size;
  *endwhile;
*endif;

*if (action = #2); RCV MSG
  Add message start delimiter to task's current history
  Add every trace elt of incoming msg to task's current history
  Add message end delimiter to task's current history
*endif;

*if (action = #3); DATA WRITE
  *pop taskstk, data_name;
  *addq trace_elt, curr_tcb.trc_q, #1;
  *move curr_tcb.trc_q, @data_name.trc_q;
*endif;

*if (action = #4); DATA READ
  Add start read delimiter to task's current history
  Append trace list of data item to task's current history
  Add end read delimiter to task's current history
*endif;

*if (action = #5); RESPONSE OUT
  Add response out trace element to task's current history
  Print (but don't delete) current history to stats file
*endif;

*if (action = #6); EVENT TRIGGER
  *pop taskstk, trace_elt.misc1;
  *etime trace_elt.misc1, trace_elt.misc2;
  *addq trace_elt, curr_tcb.trc_q, #1;
*endif;
ENDPROC;

Figure 5.22: Trace Maintenance Procedure Code
data write, the new trace element is added to the task's current working history list. The current history list of the shared data item is discarded and replaced with a copy of the writing task's current working history list. The case of data read is exactly the same as that for message receive except the history list of the read data remains unaltered because the data was not modified by the read operation.

Every event-response trace starts with the occurrence of an environmental event, usually in the form of an interrupt. When such an occurrence happens, it is up to the interrupt handler primitive to notify the task or tasks involved in initiating the response for the event. This notification can occur in several ways depending on how it is programmed by the user. The interrupt handler may simply ready the required task(s) if it was blocked waiting for the event, it may send a message to the initiating task(s), or it may update a data item signalling that the event has occurred. Each of these approaches requires different actions on the part of the response.update procedure. In each case, the procedure is called with action type 6 (event trigger) and in all cases a new trace element is created. This element contains the node name but not the task name. It also contains the interrupt number in the misc1 field and the time that the event was handled in the time field. Further, the misc2 field is set to contain the time at which the event actually occurred. This information is obtained through the etime instruction.

Once the trace element has been created, its fate is determined by the notification method. If a task (or tasks) is simply readied (made ready to run) by the event occurrence, the trace element is added to the new current working history list(s) of the readied task(s). It is necessary that the interrupt handler temporarily make the readied task(s) the current task(s) for the duration of the response.update procedure call. The procedure is called once for every readied task. If notification is through messages, the trace element becomes the history of the message and is treated as any other message send. Similarly if notification is through data, the trace element is made to be the new history list of the data item and is treated as a data write.

When a response is to be generated, the response.update procedure is called with action type 5 (response out). This causes a new trace element to be added to the
task's current working history list containing the type of response generated. Once
the history list is complete, the procedure executes the printf instruction with the
history list as the operand\(^7\). This effectively signals the end of a particular event
response trace. It is possible that this response is only one element of a sequence of
responses to be triggered by a single event. In this case, the task retains its working
history for the remaining responses.

In order to execute the above described trace actions, the following seven kernel
primitive functions can be defined to call the response_update procedure as part of
their operations.

1. **Send Message** - Sets up message sent - action (1).

2. **Receive Message** - Sets up message received - action (2).

3. **Write Shared Data** - Sets up data write - action (3).

4. **Read Shared Data** - Sets up data read - action (4).

5. **Get/Await New Environmental Event** - Sets up event trigger - action (6).

6. **Signal Response** - Sets up the response out - action (5).

7. **Start New Response** - This primitive should clear the calling task's current
   working history list when processing is started on a new response.

The response_update procedure does not dictate in any way how these primitives
should be designed (blocking vs. nonblocking, etc.) nor does it dictate the number
of primitives. Any of the specified primitive functions may be combined with others
or omitted if not needed in the application (for example, no need for message based
primitives if only shared memory communications are used). The only requirement is
that if these functions are included, they should call the response_update procedure as
well as any other user defined statistic gathering functions at the proper times with
the proper arguments.

\(^7\)The printf instruction accepts any data type as an operand and prints the value of the operand
to the statistics file. If the operand is a stack or a queue, each element is printed to the file.
Armed with the proper tools for gathering the required statistics, it is now possible to define a design level methodology that puts into practice the ideas discussed thus far in this thesis. The next chapter describes how the statistics acquired by the MDA tool are used to create a design method that addresses the decisions that must be made at the design phase.
Chapter 6

The Design Method

This chapter provides a detailed outline of the proposed design phase methodology. The previous chapters have outlined the types of decisions that must be made in transforming a functional specification to a workable design. They have also discussed the types of statistics needed to make these decisions and have described the MDA tool that is used to collect these statistics. In this chapter, these ideas are combined and put into practice in the formulation of a design methodology for the translation of a functional specification into an implementation ready design.

As previously outlined, the design options available at this phase of the development cycle can be divided into four categories: software partitioning, hardware platform, IPC scheme, and response scheduling. A number of decisions need to be made in each of these categories and in many cases, the effects of one decision will have an influence on many of the others. This implies that after each decision is made, it may render other previously made decisions inappropriate. This would require re-examination of all previous decisions after each new decision was made. Clearly this type of method leads to a combinatorial explosion in the decision making process.

In order to avoid this scenario, it is necessary to define a sequential decision making process in which there is no *haphazard* backtracking. In this process a decision, once made, should mainly influence the outcomes of subsequent decisions and have as little effect as possible on previously made decisions. Therefore, fundamental decisions must be identified and made prior to those that are more independent in nature. The
order in which the classes of decisions should be addressed during the design phase is
the same as that presented in the System Design Aspects chapter of this thesis. That
is:

1. Software partitioning

2. Selection of a hardware platform

3. Determining the nature of required IPC mechanisms

4. Selection of scheduling method

Software partitioning is the basis of the design process as it eventually determines
the number and types of processors required, the communication requirements of the
system, and also how the software components need to be scheduled. As such, this
class of decisions is the first to be made. Likewise, the hardware platform influences
the selection of interprocessor communication mechanisms based on the task distrin-
butions among the processors and the type of interlink scheme. The task to processor
allocation also effects the amount of load balancing possible by the scheduling scheme.
The third class of decisions, IPC selection, involves selecting communications meth-
ods such as shared memory or message passing, blocking or nonblocking messaging,
etc. As such, it has a direct influence on how tasks are to be scheduled to make
the most of the IPC scheme. Finally, the type of scheduling method chosen is based
largely on the previous decisions. As such, it should be the final consideration of the
design phase.

It may be argued that this type of ordering appears to be too rigid or artificial
in the sense that some later decisions may have an effect on earlier decisions and
this method does not allow enough feedback in the design loop. However it must
be stressed that this type of design is an iterative process. At some stage it may
become apparent that the chosen path is inappropriate. Under these circumstances,
it is possible to step back and revise some earlier decisions. This revision should open
up new paths which are more appropriate to the required solution. This controlled
backtracking removes the need to experiment with every possible combination of
decisions.
In complex systems that require designing via prototyping, there is not a single unique acceptable solution but rather a number of possible solutions. The idea of prototyping is to experiment with various design decisions in a systematic manner until a combination of decisions is found that satisfies the requirements of the design.

Keeping this process in mind, a design method can now be outlined by an examination of each of the four design level classes. The remainder of this chapter presents a prototyping design methodology for each individual decision class. This methodology makes extensive use of examples showing how the MDA tool can be instrumental in the progression of the design. In order to provide some level of continuity between the different decision classes and to explore a complete design sequence, the same example system will be used in demonstrating the process for all classes of decisions. The only exception is the software partitioning class. Because the chosen example is reasonably complex (as required for proper demonstration of the hardware platform and scheduling classes), it is unnecessarily involved for demonstration of the software partitioning process. This does not imply that the process can not be used on complex systems, but only that for the purpose of this thesis, a smaller example will suffice.

Prior to discussing the details of the design methodology, it is necessary to introduce the functional requirements of the example system that will serve as the target of the described process:

6.1 Requirements of a Telephone Switching Node

The chosen example focuses on some of the operations of a single switch in a telephone switching network. The layout of a typical telephone switch [Woo83, PW83] is given in figure 6.1. In this layout, the individual telephones (terminals) are connected to access nodes, the access nodes report to higher level collector nodes, and the collector nodes interface to a network that connects all of the collector nodes (and the switching center) in the switch. The switching center is the highest level of control that manages

\footnote{This example was chosen because of the author's familiarity with such systems and the ability of the example to incorporate a number of real time requirements. Although the example follows the basic philosophies of switching systems, many of their characteristics have been simplified. Also, the time values used in the example have been chosen for convenience.}
the interactions of all the lower level nodes. There is a concentration of functionality at each level of the switch. As such, there are many terminals connected to each access node, several access nodes reporting to a single collector node and all collector nodes reporting to the single switching center.

In addition to connections to higher and lower level nodes within its own switch, a collector node may also have T1 carriers (trunks) linking it with collector nodes belonging to other switches in the network\(^2\). Thus a telephone call between terminals on the same switch may require the following actions:

1. Access node detects an off-hook signal on one of its phones,

2. Access node performs digit collection to get dialled number,

3. Access node informs collector node of above actions,

4. Collector node asks switching center for a network connection,

---

\(^2\)This network links individual switches in a telephone network and is not to be confused with the network block in the switch diagram that simply links collector nodes within a single switch.
5. Collector node connects a voice channel from access node to the network.

6. Switching center informs terminating collector node of incoming call and provides it a network voice connection.

7. Terminating collector node connects network to appropriate access node and informs access node to ring appropriate telephone.

8. Once terminating access node detects off-hook, connection is complete.

9. Originating access node scans telephone for on-hook. Once received, the connection is taken down in the same order as it was made.

The actions required for a call between telephones on different switches is similar except that the originating switching center signals the terminating switching center through a trunk. The voice channel is also connected through the trunk once the call setup is complete.

6.1.1 Collector Node Hardware

To keep the example to a manageable size, only the functions of the collector node in the above system will be analyzed. Further, it is assumed that the communications between the various nodes (access nodes, collector nodes, and switching center) all take place via asynchronous communications channels (not via shared memory) and that these channels are error free. However there are no predefined restrictions concerning communications schemes within the individual nodes.

To perform all of its functions, the collector node requires some specialized hardware. A possible hardware block diagram for a collector node is given in figure 6.2. There are three main aspects to the hardware structure: the interfaces, the peripheral hardware, and the processing complex.

There are three different types of interfaces for the collector node. The network interface formats data for transmission to the network and converts incoming data from the network. The access interface performs these same functions for data to

3It is assumed that only the network block has switching capability.
and from the access nodes. Finally, the *T1 interface* hardware performs conversions between the carrier format and the internal data format of the collector node.

The functions of the peripheral hardware are quite varied and are concentrated in four hardware modules: the *signalling interface*, the *messaging interface*, *tones hardware*, and *pulse code modulation (PCM) switching*. Whenever a voice connection is set up, conversation flows through the switch in the form of PCM. The main functions of the signalling interface are to continuously monitor the quality of the voice connection and to allow the nodes involved in the connection to signal to each other over the voice channel for the duration of the call. This type of signalling involves informing nodes about things such as call setup and takedown. The signalling is done by injecting message bytes into the voice PCM stream one bit at a time (perhaps one bit over each PCM frame). In this manner, the same channel can be shared by voice and signalling data. The signalling interface simply injects and extracts these bits from the PCM stream thereby sending and receiving message bytes, respectively. For the duration of the call, test patterns are sent as signalling bytes to monitor channel integrity.

The messaging interface simply performs parallel to serial conversion on outgoing
internode message bytes and serial to parallel conversion on the incoming message bytes. It also determines which of the interface cards is appropriate for transmitting and receiving particular messages. The tones hardware provides a variety of tones (ex. dial tone, busy tone) in the form of PCM to the access nodes for switching onto appropriate terminals. Finally the PCM switch serves to connect any access channel to any network channel and ensures that the proper voice PCM bytes are switched onto the proper channels at the proper times.

Each of the peripheral hardware modules is accessible to the processing complex. The processing complex consists of the processor(s) and memory required to control the operations of the collector node. In this example, the interface and peripheral hardware is fixed and it is the design of the processing complex that will be investigated as part of the design process. The peripheral hardware and the limited number of voice channel connections are considered to be hardware resources that need to be shared between the software components of the processing complex.

### 6.1.2 Collector Node Functionality

The collector node needs to perform several functions. It must respond to environmental events as signalled by the lower level access nodes and the higher level switching center. Some of these events are periodic and others are sporadic. In addition, there are internal events that the collector node must service. In order to organize all of the various requirements, it is useful to categorize them into two classes: maintenance functions, and call processing functions.

Maintenance functions involve things such as performing diagnostics and hardware audits, synchronization with the network, making switching connections, and ensuring the integrity of any attached trunks. The required maintenance type actions are summarized in the following list.

1. Whenever a request for a tone comes from an access node, the required tone is to be connected to the specified access channel.

2. The collector node is to stay in clock synchronization with a signal received
from the network on one of the network channels. The processing complex must regularly switch the synchronization source between the network channels. As such, it must specify to the message interface which channel has been currently selected for this purpose. If synchronization is lost at any time, the messaging interface hardware generates an interrupt to the processor complex.

3. Hardware diagnostic audits are to run in the collector node. A tone audit is to run every 30 time units, a message interface audit every 5 time units, and a channel audit every 1 time unit.

4. In addition to the periodic audits, the tone diagnostics, message interface diagnostic, channel diagnostic, and a carrier diagnostic should be able to be run upon request from the switching center. Only diagnostic fail results are to be reported to the switching center. All audits and all diagnostics require two channels in order to form test looparounds.

5. The tone diagnostic must connect a test tone to a channel and monitor the tone through a second channel.

6. The channel diagnostic and the message interface diagnostic need to shut off signalling functions on the channels they are testing.

7. All T1 carriers terminating at the collector node must be continually monitored via status and control registers on the T1 interface hardware.

8. The carrier diagnostic must be able to shut off monitoring of the carrier under test.

9. The processing complex must provide signalling or test bytes to the signalling interface for each busy channel. The signalling interface hardware interrupts the processing complex whenever a complete signalling message has been received or when it is ready to accept the next message for transmission over the voice channel. This interrupt occurs at a fixed interval of 1 time unit.
10. If ever a diagnostic fails, an audit fails, a carrier is determined to be faulty, if synchronization is lost, or if the quality of a voice channel degrades, the collector node is to save all applicable data (call processing and carrier data) and immediately inform the switching center.

11. All communication between the collector node and its access nodes as well as between the collector node and the switching center is done through the message interface hardware module. During message transmission, this module interrupts the processing complex every 125 microseconds. During this time, the processing complex must put an outgoing message byte into the module’s out buffer and must retrieve an incoming message byte from the module’s in buffer. Failure to service this interrupt will result in loss of message bytes. There are 32 bytes per message.

Similarly, a number of functions related to call processing must be performed by the collector node. These functions must always take precedence over any routine maintenance actions as the goal is to able to handle as many call originsations (setups) as possible per period of time. The call processing functions are itemized below.

1. A request from the switching center or access node to setup a call requires the collector node to choose a voice channel through the node and then connect that channel to the specified path.

2. Once a connection has been setup, the signalling interface must be notified to start monitoring the integrity of the connection using specified test signalling messages. Also the state of each “in use” path must be maintained. If a problem occurs in the collector node, this is part of the data that must be stored prior to notification of the switching center. (See point 10 above).

3. When a call is to be taken down, the signalling interface collects a takedown message. Once this message is received, all connections must be broken and the channel freed.

4. The system should be able to handle several originsations concurrently.
<table>
<thead>
<tr>
<th>EVENT</th>
<th>RESPONSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Synch</td>
<td>Change Synch Source</td>
</tr>
<tr>
<td>Synch Loss</td>
<td>Report Synch Loss</td>
</tr>
<tr>
<td>Call Start Msg</td>
<td>Initiate Call Setup</td>
</tr>
<tr>
<td>Connect Tone Msg</td>
<td>Connect Tone</td>
</tr>
<tr>
<td>Diag Request Msg</td>
<td>Diagnostic Pass</td>
</tr>
<tr>
<td>Diag Request Msg</td>
<td>Report Diag Fail</td>
</tr>
<tr>
<td>Integrity Fail</td>
<td>Report Failure</td>
</tr>
<tr>
<td>Call Hung Up</td>
<td>Call Takedown</td>
</tr>
<tr>
<td>Audit Request</td>
<td>Audit Passed</td>
</tr>
<tr>
<td>Audit Request</td>
<td>Report Failure</td>
</tr>
</tbody>
</table>

Table 6.1: System Event - Response Summary

The individual input events of the collector node and their corresponding responses are given in table 6.1.

This is an extremely simplified and not completely accurate view of such a telephone controller but it does serve as a good example for illustrating the concept of making design decision through prototyping. Because this example contains many activities which can be partitioned into tasks in a number of ways, it would be an unnecessarily long process to explore the system partitioning process using this example. Instead, the next section uses a simple activity precedence graph to illustrate software partitioning. However, the telephony example will serve as the system to be modelled for all subsequent classes of design decisions.

6.2 Software Partitioning Analysis

The method of partitioning an application into software tasks is perhaps the most important aspect of the design phase as it also partly determines the fate of the later classes of decisions. To demonstrate the process of partitioning a system into tasks, the precedence graph shown in figure 6.3 will be used. This graph shows a conceptual system which is composed of five activities: A1, A2, A3, A4, and A5. A trigerring event S causes activity A4 to be started along with either activity A1 or A2. If the system condition (b) is such that b=0, then activity A1 is done otherwise A2 is done.
The completion of either A1 or A2 and A4 causes A3 to start. The completion of A4 also causes A5 to start. Once both A5 and A3 have completed, the response is completed. In addition to this precedence relation, there are specific data transfer requirements between the activities. These are specified by the labelled arrows on the connecting arcs.

In order to group the individual activities of this system into tasks through the use of prototyping, it first necessary to set up a fixed environment with respect to hardware platform, IPC methods, and scheduling. If these aspects are held constant throughout the prototyping activity, then any variations in results can be attributed to the changes made in the activity to task grouping. For this process, a single processor using asynchronous messaging will be used. The data transfers are done using messages that are all of a fixed size of 1 Kbyte (the chosen value is inconsequential as the relative numbers of message transfers remain the same regardless of the chosen size). In order to prevent any scheduling bias, a non-timesliced static priority scheduler is used throughout this section with all tasks (except the initial event servicing task) having the same priority. The initial event service task will have higher priority to minimize the number of missed events. In addition the events will be latched rather than being volatile. The code for the kernel used to implement this functionality is provided in section B.1.2.

The software partitioning process can now be described as a series of sequential operations.

**STEP 1 - Determining Data Dependencies**

The first step in the process is to determine the data and precedence relationships
that exist between the activities. This type of information shows what activities can be performed in parallel, which are sequentially dependent on one another, how much data needs to be passed between which activities, etc. These are the fundamental aspects that must be analyzed in making the proper partitioning decisions.

If the system to be designed is quite small, it is often possible to simply draw an activity precedence graph and thereby have available all of the relevant information. In the majority of cases however, it becomes an extremely arduous if not impossible task to trace through all of the interrelationships by hand. The situation becomes even more difficult if different designers are working on different parts of a large system. In such cases, the individual activities can be modelled using the MDA tool and all of the dependencies can be determined from the execution path traces. To get the absolute relationships between individual activities, it is necessary to model one activity per task during the collection of the traces. However, this fine level of detail is unnecessary because in most cases, the designer wants to deal with groups of activities right from the start; as for some activities, it may be apparent from their resource requirements which of them must be grouped in the same task.

In the example specified by the precedence graph, this modelling process is unnecessary. However, to demonstrate the process, each activity has been put into a separate task and a trace has been taken. The task diagram along with the corresponding trace is shown in figure 6.4. It should be noted that the trace provides the same information as that given by the precedence graph. In this case, a single message is sent between the tasks whenever data needs to be transferred. The contents of the message specify the amount of data involved in the transfer and can be printed if required.

The results show that the processing of a single event from the time that it was raised to the time that a response was produced, required 1218 units of time and 23 context switches (assuming each activity takes 200 units to execute). The number of context switches includes those required by the kernel to initialize the system. These will become insignificant as the number of serviced events increases.

As an aside, it should be noted that as the number of tasks are reduced, the number
Figure 6.4: System Partitioning to Determine Dependencies
of context switches and the number of messages required to produce a response is also reduced. As a result, the time required to produce the response will be lessened as well, but only up to a point. As more and more functionality is packed into fewer tasks, system concurrency is reduced so at some point, the benefits gained from reducing OS overhead (in the form of context switches and message passing) will not make up for the loss of concurrency.

STEP 2 - Initial Task Grouping

Once the data and precedence relations between the system's activities (or groups of activities) have been specified, they must be analyzed to realize the goals of the system partitioning process. These goals are: attempt to minimize the amount of intertask communication, maximize the possible concurrences, and evenly distribute CPU time requirements among the tasks. Based on the decisions described in section 4.1.1, an initial task grouping can be performed.

In order to minimize the amount of intertask communication in the given example, it may be desirable to place activities A4 and A3 in the same task as they need to exchange a large amount of data with each other. All of the other data transfers are relatively small in comparison. In order to maximize the possible concurrences, it would be ideal if the remaining activities are each put into their own tasks. However, it should be kept in mind that as the number of tasks increases, so does the system overhead. From the precedence diagram, it is observed that for any one event, only one of activities A1 or A2 will be performed. Hence there is no need to execute these in parallel and as such, they should be put in the same task. With respect to the time required by each task to service an event instance, it will be initially assumed that all the activities require the same amount of time (200 time units). From this rudimentary analysis, an initial task grouping can be obtained as illustrated in figure 6.5.

STEP 3 - Gathering Statistics

The third step in the process is to build a prototype of the proposed system and collect statistical indicators that provide concrete measures on the degree of success of the proposed partitioning scheme. The software for this first application prototype
is provided in section B.1.4. The complete set of software partitioning statistical indicators has been described in section 4.2.1. The indicators relevant to the given example along with their values for the initial task grouping are shown in table 6.2 as grouping 1. All of the simulations in this section are done for 30000 time units. The minimum response time identifies the lowest response time for the occurrence of a single event. The average number of context switches is obtained by dividing the total number of context switches by the number of responses produced. The minimum task cycle time refers to the time required by a task to service a response if there was no blocking in the cycle whereas the maximum cycle time incorporates the time spent in blocked states during a cycle. Again, in all cases, the task initially servicing the event is given higher priority.

The initial grouping was prototyped with all activities requiring 200 units of execution time. The results were collected using sporadic event arrivals with probability of occurrence of 10%. This same event arrival rate was used for all simulations in this section.

The main point that becomes evident from these statistics is that task T2 is a bottleneck for task T3. This is apparent from several of the indicators. Firstly, the average sizes of the message queues between tasks T1 and T2 are very much larger than those between T2 and T3. As a result, the time a message element spends in T2's receive queues is much longer than the time spent in T3's queues. Secondly, although the time required to complete the servicing of one set of incoming messages
<table>
<thead>
<tr>
<th>Grouping</th>
<th>No. of Completed Responses</th>
<th>Minimum Response Time</th>
<th>OS Overhead %</th>
<th>Avge. Context Switches /Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td>1015</td>
<td>16</td>
<td>9.2</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>1110</td>
<td>18</td>
<td>7.8</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>950</td>
<td>16</td>
<td>6.6</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>947</td>
<td>33</td>
<td>6.7</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
<td>407</td>
<td>20</td>
<td>3.2</td>
</tr>
<tr>
<td>Grouping</td>
<td>Avge. RTR Queue Size</td>
<td>Max. Time Spent in Message Queue</td>
<td>Average Message Queue Size</td>
<td>No. of Messages Sent</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------</td>
<td>------------------------</td>
<td>-----------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Q1 - 8208</td>
<td>Q1 - 12</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Q2 - 8217</td>
<td>Q2 - 12</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Q3 - 4624</td>
<td>Q3 - 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Q4 - 4615</td>
<td>Q4 - 1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Q1 - 7867</td>
<td>Q1 - 5</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Q2 - 8146</td>
<td>Q2 - 5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Q3 - 39</td>
<td>Q3 - 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Q4 - 32</td>
<td>Q4 - 0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>Q1 - 13683</td>
<td>Q1 - 15</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Q2 - 13485</td>
<td>Q2 - 15</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>Q1 - 1404</td>
<td>Q1 - 2</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Q2 - 1326</td>
<td>Q2 - 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Q3 - 39</td>
<td>Q3 - 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Q4 - 32</td>
<td>Q4 - 0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>Q1 - 5335</td>
<td>Q1 - 9</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Q2 - 5139</td>
<td>Q2 - 9</td>
<td></td>
</tr>
<tr>
<td>Grouping</td>
<td>Total Blocked Time/Task</td>
<td>Maximum Task Cycle Time</td>
<td>Minimum Task Cycle Time</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------</td>
<td>------------------------</td>
<td>------------------------</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>T1 - 8932</td>
<td>T1 - 997</td>
<td>T1 - 250</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T2 - 165</td>
<td>T2 - 466</td>
<td>T2 - 420</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T3 - 3001</td>
<td>T3 - 2985</td>
<td>T3 - 230</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>T1 - 7402</td>
<td>T1 - 608</td>
<td>T1 - 250</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T2 - 259</td>
<td>T2 - 808</td>
<td>T2 - 420</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>T1 - 8409</td>
<td>T1 - 754</td>
<td>T1 - 249</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T2 - 130</td>
<td>T2 - 903</td>
<td>T2 - 678</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>T1 - 5557</td>
<td>T1 - 324</td>
<td>T1 - 250</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T2 - 280</td>
<td>T2 - 594</td>
<td>T2 - 40</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>T1 - 3930</td>
<td>T1 - 393</td>
<td>T1 - 250</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T2 - 163</td>
<td>T2 - 105</td>
<td>T3 - 82</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Statistical Results of Various Groupings

178
for task T3 is only 230 units, its actual cycle time including blocking delays is 2985 units. In contrast, T2 has a service time of 420 units and its actual cycle time is 466 units. This implies that T2 is almost constantly busy catching up while T3 is very often blocked for lack of work. Finally, the average time spent blocked waiting for incoming messages or events shows that T3 and T1 are often waiting for work.

**STEP 4 - Analysis of Statistics**

From the statistics gathered during the initial prototyping, it is possible to perform a regrouping in hopes of improving the system partition. In reducing the number of tasks to three, the prototype showed improvement in the number of context switches performed. Because the three tasks do not have an equally distributed workload at this point, it may be possible and desirable to combine some of the tasks. It is apparent that activities A3 and A4 should remain in the same task to reduce the amount of IPC, but what about activity A5? Task T3 spends most of its time waiting for T2 whereas T1 spends much of its time waiting for an event occurrence. This suggests combining task T3 with task T1 as shown in figure 6.6. The application code for this grouping is provided in section B.1.5.
This grouping removes the overhead of a third task and will hopefully reduce the amount of time each task spends in the blocked state. It should be noted that adding A3 to task T1 increases the possibility that T1 will be busy when an event is raised. However this is of little concern as the events are latched and the event servicing task has high priority.

**STEP 5 - Iteration Step**

This is the iterative step in the application partitioning process. If it was decided in the previous step that a regrouping was necessary, it is necessary to go back to step 3 and repeat the prototyping activity with the new regrouping. This iterative process continues until a partitioning is found that yields acceptable statistics.

For the precedence relation example, the new regrouping yields the statistics in grouping 2 of table 6.2. This simulation was run for the same time period as the initial simulation. There are several interesting things of note in the results.

The main reason for the regrouping was to reduce the amount of time the task T3 spent blocked waiting for messages from T2 and to reduce the amount of time that task T1 spent blocked waiting for event occurrences. The results show that this objective was indeed realized. Task T1 now performs two independent functions: servicing events through activities A1 or A2, and completing responses through activity A5. In the first prototype, these two operations in separate tasks required the tasks to spend a total of 11933 (8932 + 3001) time units in the blocked state. Now, task T1 spends only 7492 units in the blocked state. In addition, the maximum combined cycle time for T1 and T3 has been reduced from 997 + 2985 units to 608 units. This is because task T1 now waits for two types of triggers and it likelihood of being triggered has been increased. Because T1 now performs more functions, it does not flood T2's message queues with requests. The average lengths of these queues have been reduced from 12 to 5.

Another effect of placing activity A5 in the same task as the event servicing activities, is that it raises the priority of A5 so that response completions are performed at a higher rate. This result is echoed in the time that a message spends in the response queues (Q3 and Q4). The time spent in these queues has been reduced from 4624
units down to 32 units. As a result, more responses are completed in the same time period (13 versus 9).

The final aspect of the new grouping is that there is one less task in the system. The results of this change are not startling in the system at hand. However, the average number of contexts switches has been reduced from 9.2 to 7.8 although the operating system overhead remains about the same. The reason for this constancy is that a complete response still must essentially go through three tasks although two of them are grouped together. This fact is also reflected in the minimum response time which is similar to the initial grouping. Because on a uniprocessor implementation, there is no true concurrency, this value will depend only on the number and types of task interactions (via messaging or other primitives). Since there is essentially no change in these types of interactions from the first grouping to this one, there is little change in the response time indicator. However, it will become significant when multiple processors are considered.

The improvements noted in this second grouping are due mainly to evenly distributing the workload among all of the tasks. That is, by giving them similar granularities. However, in hopes of improving response time by reducing the amount of required IPC, one might try the partitioning scheme given in figure 6.7. The application code for this grouping is provided in section B.1.6. In this grouping, equal task
granularities have been forsaken for a reduction in the number of messages required. The results of this grouping are shown as grouping 3 of table 6.2.

Again, there are some noteworthy aspects to these results. The first is that the number of responses produced in the 30000 units was more than the initial grouping but less than the second grouping. However, there was significant improvement in several areas over the second grouping. First of all, the number of messages being passed in the system has been reduced as there are now only two message queues. Also, the average number of context switches has been reduced because each response is handled by two tasks instead of three as in the first and second groupings. As a result, the minimum response time has been significantly lowered to 950 units from 1110. Why then, are there fewer responses produced than in the second grouping? The answer lies in task granularity.

Task T2 now requires at least 676 time units to cycle through one response whereas T1 remains at about 250. As a result, T2 is even more of a bottleneck than in the initial grouping. This is reflected in the average size of T2's incoming message queues (15) which is even larger than in the case of the first grouping (12). As a further consequence requests for T2 spend large amounts of time in the request queue (about 13000 units maximum).

In this case, the decision on the best grouping would seem to favor the second partitioning. However, this is very application dependent and partitioning 2 is the best grouping for the given messaging primitives and the given activity times with the modelled event arrival rate. If the event arrival rate was slow or sporadic enough to allow task T2 to periodically clear its request backlog, the third grouping may produce better average response times because of the fewer message interactions.

Similarly if the activities A3, A4, and A5 were of a much shorter duration than activity A1 or A2 or if the time required for message primitive execution was similar to that for activity execution, then the third grouping would likely be the best. In this case, T2 would no longer be a bottleneck as its cycle time would be reduced and it would have an advantage over the second grouping in that the now relatively expensive messaging operations would create a large overhead in the second grouping.
The above analysis is borne out by the results obtained by having activities take only 10 units of time instead of 200. Grouping 4 of table 6.2 gives the results for the grouping of figure 6.6 using these activity times while the fifth row gives the results for the grouping of figure 6.7. Also, the average number of context switches is reduced in grouping 5 as now task T2 can process responses faster and so the chances of it being preempted by an event occurrence are reduced. Clearly, all indicators point to the third grouping being the best under these activity times.

It is possible to try many different system partitioning schemes but the statistics gathered at each iteration help to focus on specific areas that need improvement. It is possible to carry out further experiments varying things such as event arrival rates and messaging overheads but the purpose of this chapter is not to provide exhaustive solutions, but only to describe a methodology for using the MDA tool to gather the described statistics for making the discussed design phase decisions.

Now that the software partitioning part of the design method has been considered, the telephony example can be revisited and a possible partitioning of that problem using the methods illustrated above can be described. Because the method has already been described, only the final result of the partitioning process of the collector node is presented.

### 6.2.1 Collector Node Software Partitioning

In order to achieve the collector node functionality described in section 6.1.2, the software has been divided into 17 tasks. The structure diagram for these tasks is shown in figure 6.8. This section very briefly describes each task. The MDA code used to model these tasks is provided in appendix B.

The message task (MSG) services a sporadic event representing the arrival of a message into the collector node by waiting for this event to be generated by the interrupt handling primitive. Once such an event occurs, the task cycles 32 times collecting the individual bytes of the message (msg size is 32 bytes). The message can be from either the switching center or the access node. A global variable msgtype
Figure 6.8: Collector Node Task Structure Diagram
is used to specify any one of the following three messages:

1. Setup a call - Communicate to CINIT task.

2. Provide a tone - Communicate to TONE task.

3. Run a diagnostic - Communicate to DIAGS task.

In addition, the MSG task accepts fault report messages from the FAULT task to be sent to the switching center. No real action is modelled for these outgoing responses except the completion of a failure response. The MSG task also accepts messages from the SYNCH task specifying how the message interface hardware is to be set for synchronization.

The synchronization task (SYNCH) waits for an event to be generated by the synch loss interrupt handler primitive. When this event occurs, communication is done with the FAULT task informing it of the failure. In addition, the SYNCH task communicates with the MSG task every 5 time units to change the channel acting as the synch source.

The TONES task accepts communications from both the MSG task (for access node requests for tone setup) and from the TONEDIAG task. Both of these requests are handled in the same way. First a fixed delay is used to simulate the selection of the proper tone and then a communication is done with the CONNECT task to connect the tone to a specified channel. It is assumed that the channel has been already acquired by the task requesting the tone operation.

The channels task (CHNLS) provides mutual exclusion for the available channels. A fixed number of channels (15) are available in the collector node. They are treated as a shared resource. The CHNLS task accepts two types of communications: *get channel* and *free channel*. The get channel operation is requested by the CINIT task and by the DIAGS task. The free channel operation is requested by the DIAGS, and END tasks when they are finished with the channels. The CHNLS task simply keeps
track of the number of free channels. The simulation doesn't actually model specific channels or their uses.

The CONNECT task accepts requests to connect and disconnect PCM channel paths. The actual operations are simply simulated as fixed time delays. It is assumed that the calling task has already secured the required channel(s). The connect and disconnect operations are used by the TONES, CHNLDIAG, CINIT, and END tasks.

The T1 maintenance task (T1MTC) performs the routine carrier function monitoring duties. These operations are simulated by fixed time delays. The occurrence of a carrier fault is modelled as an event with a probability of 10 percent. If this event occurs, the T1MTC task communicates with the FAULT task informing it of the error. This task accepts a message from the AUDITS task every second to perform its carrier monitoring duties. The T1MTC task also accepts communication from the T1DIAG task to disregard a particular carrier for some period of time. This communication causes no simulated action.

The tone diagnostic task (TONEDIAG) accepts a communication from the DIAGS task to run its diagnostic. It then communicates with the TONES task to get a tone connection. Once this is done, a fixed delay simulates the diagnostic. The probability of any diagnostic failing is 10 percent. If a failure does occur, the individual diagnostic tasks report a fail result to the FAULT task. Otherwise, a "diagnostic pass" response is completed.

The T1 diagnostic task (T1DIAG) similarly accepts a communication from the DIAGS task and then it informs the T1MTC task to stop monitoring and performs the simulated diagnostic.

The channel diagnostic task (CHNLDIAG) is invoked in the same manner. Once active, it requests a connection from the CONNECT task and then it requests that the SIGNALLING task stop its operations on a specific channel.

186
The messaging hardware diagnostic task (MSGDIAG) does the same operations as the CHNLDIAG except that it does not need to communicate with the CONNECT task.

The diagnostics task (DIAGS) accepts requests to run diagnostics from the MSG task (originating from the switching center), from the AUDITS task, and from the SIGNALLING task in the case of a voice channel integrity failure. Whenever a diagnostic request is received, this task communicates with the CHNLS task requesting two channels for use by the diagnostics. Once the channels are acquired, the DIAGS task communicates with one of the diagnostic tasks: TONEDIAG, T1DIAG, CHNLDIAG, or MSGDIAG. The choice of diagnostic selected to run is done at random.

The AUDITS task is simply a periodic task that requests diagnostics from the DIAGS task whenever the time for a particular audit comes due. As previously described, there are three such audits. The AUDITS task maintains three counters to keep the timings for these audits. The clock handling primitive causes an event in the audit task to provide it with a time base. The AUDITS task also acts as a time base for the T1MTC task by invoking it every time the timing event occurs.

The call initialization task (CINIT) accepts request from the MSG task (received from the switching center or from the access nodes) to setup a voice connection. Once this request is received, the CINIT task request a single channel from the CHANNELS task, and then requests a path connection from the CONNECT task. Once the path has been set up, it informs the STAT task that the call is now in progress.

The call status task (STAT) accepts communications from the CINIT task signifying new calls. When this occurs, it updates a call table (simulated by a fixed delay) and it requests that the SIGNALLING task start injecting test patterns into the specified channel to monitor its integrity and to scan for the call takedown message. The STAT task also accepts communications from the SIGNALLING task reporting
integrity failures. When such an occurrence happens, the STAT task informs the FAULT task and sends it one kilobyte of data to be recorded. Finally, the STAT task also accepts communications from the END task informing it that a call has been taken down. This simply causes the STAT task to update its table (fixed time simulation) and results in a “call takedown” response.

The call end task (END) accepts communications from the SIGNALLING task informing it that a caller has hung up. This causes the END task to request a disconnect operation from the CONNECT task and a free channel operation from the CHNLS task. Finally, it informs the STAT task that the call is no longer active.

The SIGNALLING task collects messages from the signalling interface hardware every second. (The interrupt handler primitive causes an event for the SIGNALLING task every second). These signalling messages can be of one of three types: integrity ok, integrity failure, or call takedown. No action is taken for integrity ok. If the message is of type integrity failure (5% probability), then the STAT task is informed and a diagnostic request is sent to the DIAGS task. If the message type is call takedown, the END task is informed. The SIGNALLING task also accepts communications from the STAT task to start integrity monitoring and from the CHNLDIAG and MSGDIAG tasks to temporarily suspend integrity monitoring. (These requests have no simulated effect).

The FAULT task accepts communications from the T1MTC, DIAGS, SYNCH, STAT, and all diagnostic tasks reporting some type of failure. If these communications come from the T1MTC or STAT tasks, there is also the need for the transfer of one kilobyte of data from the T1MTC or STAT tasks to the FAULT task. This data transfer can be simulated either by sending 30 messages or by performing numerous memory read/write operations, depending on the hardware platform. In any case, the FAULT task communicates with the MSG task to send a failure message to the switching center.
These tasks and their interactions can now be used to develop the design methodology for the hardware platform, IPC schemes, and for scheduling decisions.
6.3 Hardware Platform Analysis

Defining a hardware platform involves deciding on an appropriate number of processors needed to meet the system requirements, assigning the tasks to the individual processors, determining the number of other application specific hardware resources required, and defining a processor interlink scheme. In order to perform this type of hardware analysis through prototyping, it is necessary to have first defined the tasks that will make up the system. Secondly, it is necessary to set up a fixed environment with respect to IPC methods, scheduling, and event arrival rates. If these aspects are held constant throughout the analysis, then any variations in results can be attributed to the changes made in the hardware platform. In the examples to follow, an IPC messaging scheme using a nonblocking send combined with a blocking receive has been chosen. This protocol allows monitoring asynchronous messaging activity with respect to queue sizes, etc. and also provides some degree of synchronization. Again, as in the case of software partitioning, to prevent any scheduling bias, a non-timesliced static priority scheduler is used with all tasks having the same priority. The code for the kernel implementing this system is given in section B.2.2 and the code for the corresponding application tasks is provided in section B.2.3.

As described in section 4.1.2 prior to starting the hardware platform analysis, it is necessary to identify several system aspects such as: which tasks need to communicate with which other tasks, which tasks are server tasks that oversee data or resources, and what application specific resources need to be arbitrated. The first of these should be easily apparent after the system partitioning phase has been completed. Figure 6.8 shows the task interactions in the switching example. The identification of server tasks is done by inspecting the functionality of each task in the system. The TONES, CONNECT, and CHNLS tasks have been identified as server tasks. Finally, the only shared resources in the switching example are the channels arbitrated by the CHNLS task and shared by the diagnostic and call processing subsystems.

The first step in the hardware platform analysis is to determine the number of processors required by experimenting with several task to processor allocations. In performing an initial task to processor allocation, it is necessary to consider the
data and resource requirements of the individual tasks and their interaction patterns. Other factors such as processor contention and interprocessor communications overheads can be considered only after data has been gathered by the initial prototyping activity. In examining the task interactions in the switching example, it is evident that all of the diagnostic tasks and the AUDITS task are all dependent on one another. Similarly, the three call processing tasks (CINIT, STAT, and END) all require intercommunications. In addition, these two task groups (diagnostics and call processing) have relatively few interactions with each other. As such, the two groups can be resident on separate processors (node1 and node2 respectively). The three server tasks are used by both diagnostics and call processing so they may be resident on either processor. In the belief that call processing is functionally more important than diagnostics, the server tasks are made resident on the call processing processor (node2) in order to speed up service by eliminating the need for interprocessor communication.

This leaves the SYNCH, FAULT, SIGNAL, and MSG tasks yet to be allocated. All of these tasks interact with both subsystems so it is not obvious to which processors they should be allocated. In hopes of achieving load balancing, these tasks can be divided between the two processors. MSG and SYNCH are placed on different processors as they both service interrupts and may require significant CPU usage. The initial task to processor allocation is given in Table 6.3 along with the task specific results of this first prototyping activity. Further processor specific results are provided in Table 6.4. The tables used in this prototype are repeatedly used throughout this section. As such, their fields are described below.

The shared conflicts field of Table 6.4 shows the number of shared memory conflicts that would occur if the IPC system were implemented using shared memory. The field also provides a breakdown of the number of such conflicts associated with each receiving task. The No. of Chnls Used field gives an indication of the resource usage (in this case, the shared channels). It indicates the number of channel requests by diagnostics (MTC) and by call processing (CP). The events generated field gives the number and types of events that were raised during the simulation and the responses
<table>
<thead>
<tr>
<th>Node Name</th>
<th>Task Name</th>
<th>No. of Interprocessor Sends (Dest) (No.)</th>
<th>No. of Server Task Receptions (Src) (No.)</th>
<th>CPU Usage %</th>
<th>Time Spent in Ready State</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>msg</td>
<td>cinit (7)</td>
<td>tones (5)</td>
<td>84</td>
<td>7926</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signal (1)</td>
<td></td>
<td>3.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>audits (1)</td>
<td></td>
<td>3.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>diags (1)</td>
<td></td>
<td>3.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>tonediag (1)</td>
<td></td>
<td>3.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>chndiag (1)</td>
<td></td>
<td>3.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>msgdiag (1)</td>
<td></td>
<td>3.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>tldiag (1)</td>
<td></td>
<td>3.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>tlimtc (1)</td>
<td></td>
<td>3.4</td>
<td></td>
</tr>
<tr>
<td>node2</td>
<td>connect</td>
<td>cinit (7)</td>
<td>tones (5)</td>
<td>22</td>
<td>1198</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end (1)</td>
<td></td>
<td>3.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>chnls (1)</td>
<td></td>
<td>3.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>msg (3)</td>
<td></td>
<td>3.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>total (18)</td>
<td></td>
<td>3.4</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.3: Task Statistics for Initial Allocation

The field gives the number and types of responses that were produced. Since all events are latched and not queued, some may be lost. For this reason the event completion area gives the percentage of events that were actually serviced. The OS Overhead field gives an indication of the percentage of CPU time spent on performing system functions. The three main contributers to this number are the amount of context switching overhead, the amount of intraprocessor communication, and the amount of interprocessor communication. The idle time field indicates the percentage of free time available in the processor.

Table 6.3 gives a breakdown of the processing characteristics of the individual tasks. In this table, the interprocessor sends field gives the number of messages each task sent to other tasks not resident on the same processor. The receptions field gives the number of request messages received from other tasks. This field is used only for
<table>
<thead>
<tr>
<th>No. of Shared Conflicts Task (No.)</th>
<th>No. of Nodes</th>
<th>No. of Chnls Used (MTC/CP)</th>
<th>No. of Events Generated Type (No.)</th>
<th>No. of Responses Type (No.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg (21)</td>
<td>2</td>
<td>(0/7)</td>
<td>Change Synch (8)</td>
<td>Report Fail (3)</td>
</tr>
<tr>
<td>signal (2)</td>
<td></td>
<td></td>
<td>Synch Loss (18)</td>
<td>Conn. Tone (5)</td>
</tr>
<tr>
<td>chnl5 (7)</td>
<td></td>
<td></td>
<td>Msg In (23)</td>
<td>Call Takedown (1)</td>
</tr>
<tr>
<td>cin4it (2)</td>
<td></td>
<td></td>
<td>Signal Event (18)</td>
<td>Call Setup (7)</td>
</tr>
<tr>
<td>stat (4)</td>
<td></td>
<td></td>
<td>Audit (13)</td>
<td>Audit Pass (2)</td>
</tr>
<tr>
<td>tones (3)</td>
<td></td>
<td></td>
<td></td>
<td>23% Event Completion</td>
</tr>
<tr>
<td>TOTAL (45)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Elapsed CPU Time</th>
<th>Avge. RTR Queue Size</th>
<th>OS Overhead %</th>
<th>CPU Idle Time %</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>32771</td>
<td>1</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>node2</td>
<td>39799</td>
<td>2</td>
<td>49</td>
<td>19</td>
</tr>
</tbody>
</table>

Table 6.4: Initial Processor Allocation Results

server tasks. Finally, the *time in ready state* field indicates the number of time units each task spent in the ready to run state.

The simulation was run so that all processors run for approximately 30000 to 40000 time units\(^4\).

Several significant factors are apparent from the results obtained from the initial prototype. The most important of these is the fact that the MSG task requires over 80% of its CPU time. This was to be expected because the MSG task services all incoming messages by collecting their individual bytes one at a time. Consequently, this leaves 0% free time in node1. This heavy CPU usage by the MSG task is also apparent in the long periods of time that all tasks resident on node1 spend in the ready to run state. The inability of the MSG task to respond to all requests from outside the collector node, and the inability of the SIGNAL and AUDITS tasks to respond to all of their interrupts due to processor contention, results in a fairly low response completion percentage of 23%. For example, one of the reasons for this

\(^4\)Exact run times for individual processors can not be directly controlled but are dependent on the times at which individual processors are running locked or running in statistics gathering mode.
is that there were 13 synch loss occurrences but only one fail report response was completed. The fail report response is completed by the MSG task but this task was too busy servicing incoming message interrupts.

Further, this initial task to processor allocation results in a fairly high number of shared memory communication conflicts (45). A large number of these are due to the MSG task communicating with the call processing subsystem and to the fault task sending synch loss reports to the MSG task. For the elapsed time period, there are also a large number of interprocessor send operations (16). This serves to provide a fairly high combined OS CPU usage of 73% between the two processors. Clearly, this task to processor allocation is unacceptable and needs to be revised in the second iteration of the process.

As an aside, it is interesting to note that the diagnostic system was relatively inactive compared to call processing. In the next example, this situation is reversed. This type of bias can be eliminated by running the simulations for longer periods of time.

In the second prototyping attempt, the MSG task is given its own CPU (node 1) and all of the other tasks are made resident on node 2. There are two reasons for selecting this setup. First, it is apparent that the MSG task is instrumental in accepting messages coming from outside the collector nodes and also for completing several of the responses. Because the MSG task must run almost continuously during message reception, it requires large amounts of CPU time. Any other tasks resident on the same processor as the MSG task can potentially be held up for long periods of time. The reason for grouping all of the other tasks on node 2 is to reduce the amount of interprocessor communication and thereby reduce the amount of OS overhead as well as the number of shared memory conflicts. The results obtained from this prototype are given in tables 6.5 and 6.6.

These results show improvements in many areas over the first task allocation attempt. First of all, the goal of reducing interprocessor communication has been achieved. This is evident in the reduction of the number of shared memory conflicts (25 down from 45) and in the number of interprocessor messages (7 down from 16).
<table>
<thead>
<tr>
<th>Node Name</th>
<th>Task Name</th>
<th>No. of Interprocessor Sends (Dest) (No.)</th>
<th>No. of Server Task Receptions (Src) (No.)</th>
<th>CPU %</th>
<th>Time Spent in Ready State</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>msg</td>
<td>cinit (2)</td>
<td></td>
<td>69</td>
<td>178</td>
</tr>
<tr>
<td></td>
<td></td>
<td>diags (1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>tones (2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>node2</td>
<td>signal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>audits</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>diag</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>toneding</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>chnl.diag</td>
<td>msgdiag</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fault</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>t1diag</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>t1mnc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>connect</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>chnl</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tones</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>synch</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>cinit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>stat</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>end</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.5: Task Statistics for Second Allocation

The only interprocessor communication is now between the message task and the diagnostic and call processing subsystems. An added benefit of this allocation is that the total amount of OS overhead in the system is down from 73% to 51%. One of the reasons for this is the reduction in interprocessor communication but the major factor is the reduction in the number of tasks resident on node1. Because there is only the MSG task and the low priority idle task, there is less need for task management and context switching on node1.

It should be noted that the MSG task only required a CPU usage of 69% in this run compared to 84% in the previous run. This serves to emphasize the unpredictable nature of the incoming messages from the switching center or the access nodes to the
### Table 6.6: Second Processor Allocation Results

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Elapsed CPU Time</th>
<th>Avge. RTR Queue Size</th>
<th>OS Overhead %</th>
<th>CPU Idle Time %</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>47310</td>
<td>1</td>
<td>8</td>
<td>28</td>
</tr>
<tr>
<td>node2</td>
<td>30273</td>
<td>4</td>
<td>43</td>
<td>0</td>
</tr>
</tbody>
</table>

Collector node. To account for peak periods, it is advisable to keep the number of tasks sharing a processor with the MSG task to a minimum. However, in giving the MSG task its own processor, it seems that the other processor (node2) has been overburdened. Node2 now has 0% free time and the average size of its ready to run queue has grown to 4 tasks. While the percentage of completed responses has gone up to 28% from 23%, these response completion rates are still quite low and too many events are being missed. The reason that events are being missed is that their initial service tasks are spending too much time in the ready to run state. It is necessary to migrate one or more of these event service tasks from node2 to the lesser used node1.

The choice of task to be moved from node2 to node1 comes down to the SYNCH task, the AUDITS task, or the SIGNAL task as it is these tasks that initially service environmental events. The SYNCH task is the best choice in this case for several reasons. Firstly, on average it takes up only about 4% of the CPU time so it would not interfere with the MSG task very much. Secondly, moving this task should not adversely affect interprocessor communication as it must communicate with the MSG...
|
|---|---|---|---|---|---|
| **Node Name** | **Task Name** | **No. of Interprocessor Sends (Dest) (No.)** | **No. of Server Task Receptions (Src) (No.)** | **CPU Usage %** | **Time Spent in Ready State** |
| node1 | msg | diags (2) | 47 | 238 |
| | synch | tones (4) | | | |
| | fault | fault (1) | | | |
| node2 | signal | 9 | 8044 |
| | audits | 7.2 | 2957 |
| | diags | 8 | 7645 |
| | tonediag | 4.6 | 1839 |
| | chnldiag | 0.2 | 1730 |
| | msgdiag | 0.2 | 1784 |
| | t1diag | 6.6 | 4653 |
| | t1mtc | 4.6 | 6672 |
| | connect | tones (4) | 12.5 | 3523 |
| | end | end (1) | | | |
| chnls | 5.6 | 8131 |
| tones | diags (6) | | | |
| | end | end (1) | | | |
| cinit | 12.4 | 6333 |
| stat | mg (4) | | | |
| end | tonediag (1) | | | |
| fault | TOTAL (7) | | | | |

Table 6.7: Task Statistics for Third Allocation

Task anyway. However, it must also communicate with FAULT task so there should be little net change in the amount of interprocessor communications (unless there is a strong bias in the amount of communications towards either the MSG or FAULT tasks). The third reason for selecting the SYNCH task is that it is responsible for initial servicing of two events rather than just one (synch loss event and synch change event). As such, node2 will be relieved of two interrupts rather than one. The migration of the SYNCH task to node1 produced the results shown in tables 6.7 and 6.8.

These results are somewhat surprising in that moving the SYNCH task from node1 to node2 has resulted in increasing the free CPU time in both processors. However, the reason for the increase in free time in node1 is due to the decrease in the amount of sporadic messaging activity to the collector node from external sources as indicated
by the reduction in the number of $Msg\ In$ events. This served to limit the CPU usage by the MSG task to only 47%.

As had been suggested, the movement of the SYNCH task has had little effect on the amount of interprocessor communication and on the number of shared memory conflicts. However, the extra task in node1 has served to increase the OS overhead from 8% to 13%. The reduction in OS overhead in node2 is negligible.

The main objective in moving SYNCH from node2 to node1 was to increase the amount of free time on node2 by relieving it of some of its interrupt servicing duties. This has been accomplished but it has not resulted in an increase in the percentage of completed responses. In fact, this statistic has actually gone down from 28% to 20%. In analyzing the event data and the response data, it seems that there were a large number of synch loss events but only one fault response was sent to the switching center. The task statistics show that the FAULT task was the bottleneck in this case as it spent large amounts of time in the ready to run state on node2. It seems that there is still much CPU contention on node2. The FAULT task on average requires
Table 6.9: Task Statistics for Fourth Allocation

about 12% CPU time. In the next prototype, this task is also moved to node1 in hopes of alleviating the bottleneck. The results of this fourth allocation are shown in tables 6.9 and 6.10.

This fourth simulation highlights a number of interesting system characteristics. Moving the FAULT task to node1 has increased the amount of interprocessor communication (from 7 to 11) and consequently has increased the total amount of OS overhead in the system (although some of this increased overhead is due to the extra task being introduced into node1). In addition, in the case of synch loss, the bottleneck has been moved from the FAULT task to the SYNCH task. Now that the SYNCH task is competing for CPU access with both the MSG task and the FAULT task, it spends a large amount of time in the ready to run state. Consequently, many
<table>
<thead>
<tr>
<th>No. of Shared Conflicts, Task (No.)</th>
<th>No. of Nodes</th>
<th>No. of Chns Used (MTC/CP)</th>
<th>No. of Events Generated, Type (No.)</th>
<th>No. of Responses, Type (No.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg (9)</td>
<td>2</td>
<td>(6/3)</td>
<td>Change Synch (17)</td>
<td>Report Fail (13)</td>
</tr>
<tr>
<td>diag (8)</td>
<td></td>
<td></td>
<td>Synch Loss (36)</td>
<td>Conn. Tone (3)</td>
</tr>
<tr>
<td>fault (4)</td>
<td></td>
<td></td>
<td>Msg In (22)</td>
<td>Call Takedown (3)</td>
</tr>
<tr>
<td>tones (3)</td>
<td></td>
<td></td>
<td>Signal Event (9)</td>
<td>Call Setup (3)</td>
</tr>
<tr>
<td>cinit (7)</td>
<td></td>
<td></td>
<td>Audit (3)</td>
<td>Audit Pass (4)</td>
</tr>
<tr>
<td>TOTAL (20)</td>
<td></td>
<td></td>
<td></td>
<td>Synch Changed (3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Diag Pass (3)</td>
</tr>
</tbody>
</table>

30% Event Completion

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Elapsed CPU Time</th>
<th>Avge. RTR Queue Size</th>
<th>OS Overhead %</th>
<th>CPU Idle Time %</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>35638</td>
<td>2</td>
<td>23</td>
<td>2</td>
</tr>
<tr>
<td>node2</td>
<td>34104</td>
<td>4</td>
<td>44</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.10: Fourth Processor Allocation Results

of the synch loss events are missed.

Although many of the synch loss events are missed, more of those that are captured are resulting in completed fault response messages being sent to the switching center. This is because the FAULT task is now better able to process these responses. As a result, the percentage of completed responses has risen to 30%.

However, the results also show that both node1 and node2 are now being heavily used with 2% and 0% free time respectively. In addition, the average ready to run queue sizes are growing in both processors. The first simulation showed heavy usage of system resources by the call processing subsystem and the third simulation showed a high level of usage by the maintenance or diagnostic subsystem. In this fourth simulation, both the call processing and diagnostic systems show moderate levels of activity. Under such circumstances, the processors can not provide enough processing power to adequately meet all the needs of the system. A third processor must be introduced.

With the introduction of a third processor and the knowledge acquired from the
two processor prototypes, it is possible to perform an intelligent task to processor allocation. It has been shown that the MSG task can undergo wide variations but has the potential to use large amounts of CPU time. To prevent it from starving other tasks, it should be allocated a dedicated processor. Also, the DIAGS, AUDITS, FAULT, and SYNCH tasks are quite interdependent and it has been shown that under some conditions, these diagnostic subsystem tasks can together use large amounts of CPU time. Similarly, at times of peak call periods, the call processing subsystem (CINIT, STAT, END, and SIGNAL) are capable of monopolizing a processor. As a result, the diagnostic tasks and the call processing tasks should be placed on separate processors. The only remaining question is where to put the three service tasks (TONES, CHNLS, and CONNECT)?

To reduce the amount of interprocessor communication, it seems reasonable to place them on the same processor that generates most of their requests. The simulations thus far have shown that the diagnostic and call processing subsystems have been almost equal users of the CHNLS task, call processing has been the more frequent user of the CONNECT task, and diagnostics and MSG have been the more frequent users of the TONES task. In addition, the results have shown that the TONES task frequently uses the CONNECT task. Thus there is no clear cut indication where the service tasks should be placed.

The three service tasks require significant amounts of CPU time (collectively on average about 30%) and thus they may delay other tasks on the same processor from running. Because the call processing functions are more urgent than the diagnostic functions, they can not afford to be delayed by the service tasks. However, the channel requests for call processing are more important than channel requests for diagnostics. As a result, for the initial three processor prototype, the TONES and CONNECT service tasks will be placed in the diagnostic processor (node2) and only the CHNLS service task will be placed in the call processing processor (node3). The results of this fifth prototype are given in tables 6.11 and 6.12.

The results show that there are advantages and disadvantages in adding the third
<table>
<thead>
<tr>
<th>Node Name</th>
<th>Task Name</th>
<th>No. of Interprocessor Sends (Dest) (No.)</th>
<th>No. of Server Task Receptions (Src) (No.)</th>
<th>CPU Usage %</th>
<th>Time Spent in Ready State</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>msg</td>
<td>cinit (6)</td>
<td></td>
<td>78</td>
<td>178</td>
</tr>
<tr>
<td></td>
<td></td>
<td>diags (3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>tones (1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>node2</td>
<td>synch</td>
<td>msg (1)</td>
<td></td>
<td>3.4</td>
<td>12147</td>
</tr>
<tr>
<td>fault</td>
<td>audits</td>
<td>msg (3)</td>
<td></td>
<td>21.7</td>
<td>6941</td>
</tr>
<tr>
<td>diags</td>
<td></td>
<td>chnls (6)</td>
<td></td>
<td>3.6</td>
<td>1167</td>
</tr>
<tr>
<td>toneding</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td>8651</td>
</tr>
<tr>
<td>magdiag</td>
<td></td>
<td></td>
<td></td>
<td>5.2</td>
<td>1646</td>
</tr>
<tr>
<td>tdiag</td>
<td></td>
<td></td>
<td></td>
<td>0.2</td>
<td>546</td>
</tr>
<tr>
<td>tlimt</td>
<td></td>
<td></td>
<td></td>
<td>0.2</td>
<td>600</td>
</tr>
<tr>
<td>connect</td>
<td></td>
<td></td>
<td></td>
<td>12.2</td>
<td>1647</td>
</tr>
<tr>
<td></td>
<td></td>
<td>msg (1)</td>
<td></td>
<td>4.1</td>
<td>1686</td>
</tr>
<tr>
<td></td>
<td></td>
<td>toneding (1)</td>
<td></td>
<td>8.1</td>
<td>3796</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>22.6</td>
<td>5462</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>node3</td>
<td>chnls</td>
<td></td>
<td></td>
<td>11.6</td>
<td>3787</td>
</tr>
<tr>
<td>cinit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>signal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>connect (6)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TOTAL (26)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.11: Task Statistics for Fifth Allocation

processor. The disadvantages come in the form of increased interprocessor communication. The number of interprocessor messages has jumped to 26 and correspondingly, the number of shared memory conflicts has climbed to 89. However, these types of results are to be expected with additional processors and the hope is that they will be compensated for by the increased overall system responsiveness. The results show that this is indeed the case as now over 40% of the generated events result in completed responses. This is a significant increase over the two processor prototypes.

Now that is has been established that adding the third processor has been beneficial, an analysis should be done to determine whether the tasks have been properly allocated. The results show that node2 is perhaps being overused while node3 is being
Table 6.12: Fifth Processor Allocation Results

underused with 3% and 27% free CPU time respectively. In this case, the application may be biased towards diagnostic operations rather than call processing. In this simulation, it is the SYNCH task on node2 that suffers from the low CPU availability as it spends a large amount of time in the ready to run state. Consequently, many of the synch change and synch loss events are missed.

Also, it seems that the decision to put the CONNECT task on node2 has contributed to a significant number of interprocessor communications (6 out of a total of 26). On the other hand, the decision to put the CHNLS service task on node3 seems to have been a good one. This task received the same number of requests from both the diagnostic and call processing subsystems but its CPU usage could not have been tolerated on node2. In fact, node2 can not even adequately support the TONES and CONNECT service tasks as they are using over 30% of the CPU time. Moving the TONES and CONNECT tasks from node2 to node3 should help equalize CPU usage
<table>
<thead>
<tr>
<th>Node Name</th>
<th>Task Name</th>
<th>No. of Interprocessor Sends</th>
<th>No. of Server Task Receptions</th>
<th>CPU Usage%</th>
<th>Time Spent in Ready State</th>
</tr>
</thead>
</table>
| node1     | msg       | cinit (6)  
diags (1)  
tones (4)   |                              | 73         | 290                       |
|           |           |                             |                              |            |                           |
| node2     | synch     | msg (1)  
msg (5)  
chnls (2) |                              | 6.3        | 11246                     |
|           | fault     |                              |                              |            | 6662                      |
|           | audits    |                              |                              | 31         | 1788                      |
|           | diags     |                              |                              | 7          | 7088                      |
|           | tonedias  |                              |                              | 11.2       | 836                       |
|           | chnldias  |                              |                              | 6.4        | 523                       |
|           | msgdiag   |                              |                              | 0.5        | 667                       |
|           | tldiag    |                              |                              | 0.5        | 2815                      |
|           | timec     |                              |                              | 12.8       | 4439                      |
|           |           |                             |                              | 6.4        |                           |
| node3     | tones     | msg (4)  
tonedia (1) |                              | 8.3        | 811                       |
|           | connect   |                              |                              | 21.2       | 2777                      |
|           | chnls     |                              |                              | 3.7        | 5069                      |
|           | cinit     |                              |                              |            |                           |
|           | signal    |                              |                              |            |                           |
|           | stat      |                              |                              |            |                           |
|           | end       |                              |                              |            |                           |
|           | TOTAL     |                             | (19)                         |            |                           |

Table 6.13: Task Statistics for Sixth Allocation

(load balancing) and it should also decrease the amount of interprocessor communications. The results of this sixth allocation prototype show that this is indeed the case. The results are presented in tables 6.13 and 6.14.

This final task to processor allocation results in approximately the same responsiveness as in the previous allocation (42% of event responses completed) but the number of interprocessor communications has been reduced from 26 to 19 and there has been a more equitable load balance between the three processors as now the CPU free times are 25%, 13%, and 21% for node1, node2, and node3 respectively. While the increase in responsiveness over the previous scheme is not immediately apparent, the fact that there is a more equal CPU usage suggests that this final allocation
Table 6.14: Sixth Processor Allocation Results

scheme has the potential to handle more events than the previous scheme. This process has demonstrated a method that can be used for performing task to processor allocation and thereby choosing the appropriate number of processors for a particular application. In this example, three processors were required and a reasonable task to processor allocation was arrived at in the sixth prototype. It is interesting to note that two potential problem areas failed to materialize in the simulations.

The first potential problem could have been in deciding where to place the FAULT task. This task is used both by diagnostics and by call processing subsystems to report failures. In addition, for certain types of failures, there is a need to transfer large amounts of data between the FAULT task and the task calling the FAULT task. Ideally, the FAULT task should be placed in the processor that reports most failures.

\[\text{42\% Event Completion}\]

\[\begin{array}{|c|c|c|c|c|}
\hline
\text{Node Name} & \text{Elapsed CPU Time} & \text{Avge. RTR Queue Size} & \text{OS Overhead \%} & \text{CPU Idle Time \%} \\
\hline
\text{node1} & 35830 & 1 & 16 & 25 \\
\text{node2} & 30861 & 2 & 48 & 13 \\
\text{node3} & 38229 & 1 & 40 & 21 \\
\hline
\end{array}\]

\[\text{Although a 43\% response completion rate seems fairly low for this application, the numbers chosen for event arrival rates do not necessarily represent a real life situation.}\]
or in the processor that requires the most failure data transfers. If these requirements become conflicting then there is a problem. However, the simulation showed that the failure cases that require large data transfer are so infrequent, that this should not be a consideration in deciding the proper placement of the FAULT task\textsuperscript{6}.

The second potential problem was in the area of shared resource contention. Deciding on the proper hardware platform includes determining the proper number of shared resources and imposing proper arbitration policies on them. In this case, the shared resources were the 15 voice channels shared by diagnostics and by call processing. All of the simulations showed that 15 channels are adequate to meet the needs of the modelled system. In most cases, the system could not process enough events to warrant even 15 channels. It is possible that this system is overconfigured and if channels were an expensive resource, their number could be reduced to better match the processing capability of the system. However, these numbers are strongly influenced by implementation aspects and should be taken as guidelines only.

The other significant aspect of shared resource usage that was revealed by the simulations is that in most cases, the diagnostic subsystem used as many channels as the call processing subsystem. In the prototypes this was not a problem as there were enough channels to go around but if the number of channels is reduced, there is a possibility that the diagnostic subsystem could starve call processing of free voice channels. This highlights the need for the implementation of some type of allocation policy that would give call processing priority over diagnostics in channel allocations.

Now that task to processor allocation, number of processors, and resource usage have been considered, the remaining hardware platform issue involves the selection of a processor interlink scheme. Because the given example consists of only three processors and table 6.13 shows that all three processors must communicate with each other, there is no question about interlink topology (all three need to be interconnected). However, in other systems, the simulations may indicate that a fully interconnected scheme is not warranted.

Table 6.14 shows that if shared memory is used as the interlink medium between

\textsuperscript{6}It is important to have an accurate figure for the percentage of such fail cases in the modelled prototype. This figure could perhaps be obtained from actual field observations.
the processors, a relatively large amount of contention can be expected. The breakdown statistics involving the shared memory conflicts can be analyzed to determine the communication patterns and communication volumes between the processors. In this case, it seems that most of the contention is due to the MSG task on node1 communicating with the CINIT task on node3 and the DIAGS and TONES tasks on node2. The other area of contention seems to be between the DIAGS task on node2 and the CHNLS task on node3. Thus it may be advisable to use separate memories for communication between node1 and node2, between node1 and node3, and between node2 and node3 rather than using a single monolithic shared memory between all three processors.

If a communications link is used to handle interprocessor communications, the simulation results (interprocessor sends) can be used to determine what volumes of traffic can be expected between the various processors for a given time period. This data can be used to estimate the amount of bandwidth required between processors and can thus influence the selection of the required communications link.

The method of using prototyping to make decisions in the design phase relating to selection of hardware platform is applicable to any system. All that is required is the selection of the applicable statistics for the particular system (as outlined in section 4.1.2) followed by the iterative prototyping process described above. Once the hardware platform has been finalized, the choice of the IPC scheme can be examined.

6.4 Communication Primitives Analysis

As mentioned in section 4.1.3, it is not always possible to select or modify the choice of IPC schemes available for a particular application. As such, it is difficult to outline a fixed process that can be used for all situations. The best that can be hoped to be achieved by such a process is to be able to highlight the benefits and drawbacks of various approaches as they relate to the application in question.

In choosing the proper types of intertask communication mechanisms, there are three main considerations: identifying the needs for various IPC features, determining
the time costs and benefits of the primitives, and determining the effects of the chosen primitives on the structure of the application software. In order to perform this type of IPC analysis through prototyping, it necessary to specify a fixed environment in which the only changes involve the communication primitives themselves. To demonstrate the IPC analysis process, the telephone switching example will be used using the final software partitioning and hardware platform defined in the previous section (three processors with the task to processor allocation defined in table 6.13). In addition, a shared memory will be assumed as the processor interlink mechanism so that both shared memory and message passing IPC constructs can be considered. Finally, a fixed priority, nonpreemptive scheduling scheme will be used with all tasks having the same priority.

For the purpose of this discussion, four separate kernels were analyzed, each incorporating a different set of IPC primitives. In order to get a fair comparison, all prototypes used the same application software with only specific changes needed to adapt it to the different IPC mechanisms. The four kernels used and their IPC primitives are presented in the following list:

1. Semaphore Based Kernel - SIGNAL and WAIT operations.

2. Synchronous Message Passing Kernel - Blocking SEND with REPLY and blocking RECEIVE operations.


4. Asynchronous Message Passing Kernel - Nonblocking SEND and receive specific blocking RECEIVE.

Simulations using these four kernel prototypes were all run for 125000 units of simulated time (with each processor running for approximately 35000 time units) and all were run under the same event arrival conditions (sporadic synch loss with 5% probability, cyclic synch change, sporadic msg in with 20% probability, cyclic audits, and cyclic signalling events). The actual MDA code used to implement the kernels as
well as their respective applications is provided in section B.3. The relevant statistics gathered by each of the prototypes are presented in tables 6.15 to 6.18. In these tables, the *Shared Memory Conflicts* field represents the number of times that two or more processors made simultaneous access to data resident in shared memory for the purpose of interprocessor communication. The *OS Ovhd* field represents the percentage of CPU time used in executing inside the kernel. This value represents time spent in IPC primitive execution as well as that spent in other operating system functions. In addition, for each of the kernels, the number of times each IPC primitive was called is given along with the average execution time of that primitive. Finally, a breakdown is given with respect to the number and types of event occurrences and responses as well as a percentage of events for which complete responses were generated. This last statistic gives a good indication of the relative efficiency of the modelled system as the number and distribution of event type occurrences was similar for all simulations.
<table>
<thead>
<tr>
<th>Node Name</th>
<th>Final CPU Time</th>
<th>No. of Shared Memory Confli.</th>
<th>OS Ovhd %</th>
<th>No. of Context Switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>40237</td>
<td>3678</td>
<td>4.6</td>
<td>36</td>
</tr>
<tr>
<td>node2</td>
<td>33861</td>
<td></td>
<td>28</td>
<td>93</td>
</tr>
<tr>
<td>node3</td>
<td>41809</td>
<td></td>
<td>10</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TOTAL (182)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th># of WAIT Calls Made</th>
<th>Total WAIT Exec. Time</th>
<th># of SIGNAL Calls Made</th>
<th>Total SIGNAL Exec. Time</th>
<th>Avge. SEMA Queue Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>60</td>
<td>320</td>
<td>36</td>
<td>792</td>
<td>1.0</td>
</tr>
<tr>
<td>node2</td>
<td>134</td>
<td>736</td>
<td>150</td>
<td>9552</td>
<td>1.0</td>
</tr>
<tr>
<td>node3</td>
<td>69</td>
<td>426</td>
<td>55</td>
<td>2810</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>AVGE. /CALL 5.6</td>
<td>TOTAL 241</td>
<td>AVGE. /CALL 54.6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Avge. Data Buffer Size</th>
<th>Event Data Type (# Raised) (% Completed)</th>
<th>Response Data (# Completed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>mag (3)</td>
<td>Change Synch (8) (75)</td>
<td>No. of Responses = 32</td>
</tr>
<tr>
<td>node2</td>
<td>ALL (0)</td>
<td>Synch Loss (24) (71)</td>
<td>% Completion of Event</td>
</tr>
<tr>
<td>node3</td>
<td>chnl3 (1)</td>
<td>Mag In (32) (13)</td>
<td>Responses = 44%</td>
</tr>
<tr>
<td></td>
<td>OTHERS (0)</td>
<td>Signal Event (3) (100)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Audit (6) (100)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TOTAL (73)</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.15: Semaphore Based System Results
<table>
<thead>
<tr>
<th>Node Name</th>
<th>Final CPU Time</th>
<th>No. of Shared Memory Confl.</th>
<th>OS Ovhd %</th>
<th># of Context Switches (With Kernel Buffering)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>38028</td>
<td>91</td>
<td>21</td>
<td>68 (43)</td>
</tr>
<tr>
<td>node2</td>
<td>33002</td>
<td></td>
<td>59</td>
<td>195 (86)</td>
</tr>
<tr>
<td>node3</td>
<td>41132</td>
<td></td>
<td>43</td>
<td>183 (74)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TOT. 446 (203)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th># of SEND Calls Made</th>
<th>Total SEND Exec. Time</th>
<th># of RECV Calls Made</th>
<th>Total RECV Exec. Time</th>
<th>Total # Msgs Sent</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>9</td>
<td>1077</td>
<td>20</td>
<td>1958</td>
<td>19</td>
</tr>
<tr>
<td>node2</td>
<td>35</td>
<td>5833</td>
<td>23</td>
<td>2588</td>
<td>53</td>
</tr>
<tr>
<td>node3</td>
<td>19</td>
<td>1954</td>
<td>44</td>
<td>4758</td>
<td>53</td>
</tr>
<tr>
<td>TOTAL</td>
<td>63</td>
<td>AVGE. /CALL 141</td>
<td>87</td>
<td>AVGE. /CALL 107</td>
<td>125</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th># of REPLY Calls Made</th>
<th>Total REPLY Exec. Time</th>
<th>Event Data Type (# Raised)</th>
<th>Response Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>10</td>
<td>1012</td>
<td>Change Synch (4)</td>
<td>No. of Responses 19</td>
</tr>
<tr>
<td>node2</td>
<td>18</td>
<td>1726</td>
<td>Synch Loss (9)</td>
<td>Completed 27%</td>
</tr>
<tr>
<td>node3</td>
<td>34</td>
<td>3137</td>
<td>Msg In (47)</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>62</td>
<td>AVGE. /CALL 95</td>
<td>Signal Event (7)</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.16: Synchronous System Results
<table>
<thead>
<tr>
<th>Node Name</th>
<th>Final CPU Time</th>
<th>No. of Shared Memory Conflicts</th>
<th>OS Ovhd %</th>
<th>No. of Context Switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>39483</td>
<td>80</td>
<td>13</td>
<td>8</td>
</tr>
<tr>
<td>node2</td>
<td>32733</td>
<td></td>
<td>42</td>
<td>84</td>
</tr>
<tr>
<td>node3</td>
<td>40479</td>
<td></td>
<td>41</td>
<td>81</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TOTAL (173)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th># of SEND Calls Made</th>
<th>Total SEND Exec. Time</th>
<th># of RECV Calls Made</th>
<th>Total RECV Exec. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>24</td>
<td>2142</td>
<td>2</td>
<td>200</td>
</tr>
<tr>
<td>node2</td>
<td>51</td>
<td>5640</td>
<td>40</td>
<td>4829</td>
</tr>
<tr>
<td>node3</td>
<td>53</td>
<td>2979</td>
<td>82</td>
<td>9336</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>AVGE. /CALL</td>
<td>84</td>
<td>TOTAL AVGE. /CALL</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>124</td>
<td></td>
<td>116</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Incoming Msg Queue Size (ave, max)</th>
<th>Event Data - Type (# Raised)</th>
<th>Response Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>msg (4,11)</td>
<td>Change Synch (5)</td>
<td>No. of Responses</td>
</tr>
<tr>
<td></td>
<td>fault (1,3)</td>
<td>Synch Loss (18)</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>diags (0,1)</td>
<td>Msg In (45)</td>
<td>Completed 29%</td>
</tr>
<tr>
<td></td>
<td>t1mrc (0,1)</td>
<td>Signal Event (15)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>msgdiag (0,1)</td>
<td>Audit (3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tonediag (0,1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>chnldiag (0,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>t1diag (0,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>node2</td>
<td>cinit (0,2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>chnls (1,2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>end (0,1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>connect (0,2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>stat (0,2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>signal (0,1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tones (1,3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.17: Receive Specific Asynchronous System Results
<table>
<thead>
<tr>
<th>Node Name</th>
<th>Final CPU Time</th>
<th>No. of Shared Memory Cons.</th>
<th>OS Overhd %</th>
<th>No. of Context Switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>38140</td>
<td>97</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>node2</td>
<td>33341</td>
<td></td>
<td>39</td>
<td>71</td>
</tr>
<tr>
<td>node3</td>
<td>41874</td>
<td></td>
<td>34</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TOTAL (142)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th># of SEND Calls Made</th>
<th>Total SEND Exec. Time</th>
<th># of RECV Calls Made</th>
<th>Total RECV Exec. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>24</td>
<td>2064</td>
<td>3</td>
<td>312</td>
</tr>
<tr>
<td>node2</td>
<td>49</td>
<td>5695</td>
<td>46</td>
<td>5332</td>
</tr>
<tr>
<td>node3</td>
<td>61</td>
<td>3791</td>
<td>83</td>
<td>8712</td>
</tr>
<tr>
<td></td>
<td>TOTAL 134</td>
<td>AVGE. /CALL 86</td>
<td>TOTAL 132</td>
<td>AVGE. /CALL 109</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Incoming Msg Queue Size (avge, max)</th>
<th>Event Data Type (# Raised)</th>
<th>Response Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>mag (5,15)</td>
<td>Change Sync (5)</td>
<td></td>
</tr>
<tr>
<td>node2</td>
<td>fault (1,1) diagonal (0,1) tlmkc (0,2) msgdiag (0,0) tonediag (0,1) chmdiac (0,0) tldiag (0,1)</td>
<td>Synch Loss (13) Msg In (48) Signal Event (15) Audit (4)</td>
<td>No. of Responses 27 Completed 32%</td>
</tr>
<tr>
<td>node3</td>
<td>cinit (1,3) chlns (0,3) end (0,1) connect (0,1) stat (0,2) signal (1,3) tones (1,4)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.4.1 Semaphore Constructs vs. Messaging Constructs

The semaphore based system is the most basic of the four prototypes as it simply provides mutual exclusion primitives and allows the application software to directly perform its own data transfers as required. In a shared memory environment, the message passing systems simply provide a uniform, well-structured overlay that handle both the mutual exclusion and the data transfer. In fact, the first point that should be noted in this comparison is that in a shared memory environment, specific data transfer primitives are not essential. This is borne out by the fact that the exact same functionality has been provided in the system using semaphore operations as those using message passing.

In comparing the semaphore system with the message based systems, it is readily apparent that the semaphore kernel is the most efficient in terms of time overheads. This is confirmed by many of the statistics of table 6.15. The total operating system overhead is by far lowest in the semaphore kernel with node2 having the highest usage at 28% which is still much lower than the average overhead shown by the messaging kernels. The main reason for this kernel efficiency is in the times spent in executing the semaphore wait and signal operations. The wait operation has an average execution time of only 6 time units and the signal operation averages only 55 time units. This compares extremely favorably with the message primitives which require between 84 and 141 time units. Of course these statistics are strongly influenced by the manner in which the primitives are programmed. However, in the four prototypes, all of the primitives have been programmed in a straightforward manner in accordance with the standard definitions regarding the functionality of each of the primitives.

Because of the efficiency of the semaphore kernel, the prototype of table 6.15 was able to complete responses to 44% of the events generated. This is significantly better than the 32% of its nearest message based competitor\(^7\). From these results, it would seem that semaphore based systems should be the preferred choice in a shared memory environment. However, this system does have some significant drawbacks.

\(^7\)The event response completion results of this section should not be compared with those of other sections (such as the hardware platform analysis) as higher event arrival rates were used in this IPC section to illustrate the determination of queue lengths by prototyping.
In comparing table 6.15 with the results obtained from the message based kernels, one of the most striking statistics is the amount of shared memory contention that the semaphore based system suffers. During the simulation run, there were 3678 instances when two or more processors attempted to access shared data during the same simulation cycle. This can be attributed to the fact that it is the numerous independent application tasks (rather than a single central kernel) that are directly manipulating shared semaphores and data buffers to perform intertask and interprocessor communication.

Each such communication requires that the sending application task perform the following actions:

1. Wait for mutually exclusive access to the data area of the destination task. This is done by performing a wait operation on a mutex semaphore.

2. Once access is granted, the sending task accesses the destination data area and copies the data item to it.

3. When the copy operation is complete, the sending task performs a signal operation on the mutex semaphore to allow other tasks access.

4. Finally, the sending task must perform a signal operation on a semaphore used to determine the availability of pending request data. The receiving task will be waiting on this semaphore.

Thus for the equivalent of every send operation, an application task must invoke one wait primitive and two signal primitives. This accounts for the high number of primitive calls performed in the semaphore system as shown in table 6.15. Although the number of primitive calls is compensated by the relatively short primitive execution times, it does add significant complexity to the structure of the application tasks.

From the MDA code given in sections B.3.5 and B.3.6, it is evident that the semaphore based system has shifted complexity from the kernel software to the application software. The application tasks contain many more states than those required for the message based systems. In addition, there are many more potentially blocking
wait states in the semaphore applications than in the asynchronous messaging based tasks. This is due to the fact that every data transfer involves a call to the wait primitive in order to gain mutually exclusive access to the destination data buffer. The main implication of the increased potential for blocking is that tasks that are used to perform initial event servicing (such as the MSG task) may spend more time being blocked on semaphores and thus may miss a greater proportion of volatile or latched events.

While semaphores provide a means for very efficient communication, their use can become cumbersome for large applications involving many designers. The problem is compounded if the application requires a large amount of intertask communications. However, for smaller systems, semaphores may be the IPC method of choice as their use provides the potential for high throughput and responsiveness.

6.4.2 Synchronous vs. Asynchronous Messaging Primitives

The most profound effect of providing messaging IPC primitives such as send, receive, and reply is that the application software can take on a much simpler structure than that using semaphore operations. It can be seen from the application tasks in section B.3.4 that those using messaging primitives not only have many fewer states but they are also easier to follow logically. While the simplification of the application software is evident in both the blocking and nonblocking cases, it is most pronounced in the case of the blocking send.

Consider the three tasks CHNLS, DIAGS, and CINIT. The CHNLS task is a server task that allocates voice channels to both the main diagnostics task (DIAGS) and to the main call processing task (CINIT). Both of these client tasks send request messages to the CHNLS task to acquire a channel and then must wait for an acknowledgement message from the CHNLS task before proceeding. However, the client tasks are expected to handle incoming messages from other tasks as well. The problem arises when a client task has sent a channel request message and is waiting for an acknowledgement message. In the case of the nonblocking kernel, once the client task has sent the channel request message, it is free to continue execution.
As such, it must explicitly call the receive primitive to await the acknowledgement message. However, this receive request may be satisfied by some incoming message other than the channel acknowledgement. In this case, the client task must accept the message and queue it internally for future use. This "receive and queue" process must continue until the acknowledgement message is received.

In the case of the blocking send - reply kernel, there is no need for the application tasks to do their own internal message queue management in this manner. After the client task sends the channel request message, it is blocked until the acknowledgement message is sent. When the client task resumes execution, it implicitly knows that its channel request has been satisfied and that it is free to continue without the need for polling for acknowledgement messages. For this type of "handshaking" task interaction, a blocking send primitive becomes extremely useful.

However, there is a price to be paid for the reduced complexity of the application software using the blocking send - reply kernel. Table 6.16 shows that completed responses were provided for 27% of the events raised in the blocking system prototype. In contrast, table 6.18 shows that complete responses were provided for 32% of the generated events in the nonblocking system. The main reason for the reduced responsiveness of the blocking system is the increased complexity of the kernel and the fact that initial event servicing tasks are more often in the blocked state and thus miss a greater number of volatile and latched events.

Table 6.16 shows that the prototype using the blocking IPC kernel suffers from a total of (29% + 59% + 43%) 123 OS overhead across the three processors. In contrast, table 6.18 shows that the nonblocking system accrues a total of (11% + 39% + 34%) 84 overhead across the three processors. There are several reasons for this phenomenon. The first is that the blocking IPC primitives are computationally more expensive than their nonblocking counterparts. The average time for a blocking send operation is 141 time units while that of the nonblocking send is 86 time units. There is a much less pronounced difference in the average execution times of the receive operation as they both require approximately 108 time units. The reply operation of the blocking kernel has no counterpart in the nonblocking kernel and adds an extra
95 time unit overhead at every invocation. In the case of the blocking kernel, every data transfer operation requires a call to both the send and the reply primitives. This in effect increases the average time for a data transfer to a total of 236 time units for the blocking kernel.

Apart from the added complexity of the blocking primitives and the need for a larger number of primitive calls, the blocking nature of the primitives themselves contribute to the increased OS overhead. This is evident in the number of context switches incurred by the blocking and the nonblocking kernels. In the case of the nonblocking kernel, there are a total of 142 context switches across the three processors. In the case of the blocking kernel, there are 203 switches (provided the blocking kernel does internal message buffering). If the blocking kernel does not implement any type of kernel message buffering, this number jumps to 446. The only time a context switch occurs in the nonblocking kernel is when a task calls the receive primitive and there is no message pending. In contrast, in the synchronous kernel, every call to the blocking send primitive results in a context switch as does the call to the receive primitive when there is no message pending.

The nonblocking kernel always needs to perform message buffering but it is not necessary in the case of the blocking send. The prototype results show that the maximum size reached by any message buffer in the nonblocking case is 15 (for the MSG task). However, this seems to be an extreme case with the more common maximum queue lengths being in the 2 to 4 range. This type of statistic is quite useful in the design process in determining the amount of memory to allocate for message queues if a nonblocking environment is chosen. With the blocking environment, there are two possible approaches to buffering messages. In the first approach, there is no message buffering in the kernel. The message transfer proceeds by the kernel switching to the context of the sending task, copying the message to a temporary space, and then switching to the destination task context to complete the copy operation as part of the receive primitive. As such, each data transfer requires two context switches. An alternative is for the kernel to internally buffer a message as soon as a send operation occurs. When the corresponding receive is done, the message is ready in the kernel
and there is no need to do a context switch to the sending (source) task. If the cost of a context switch is high for a particular implementation and a blocking kernel is required, it may be beneficial to use kernel buffering.

Another aspect of the blocking environment is that it causes tasks to spend longer periods of time in the blocked state. Each time a task performs a send operation, it is blocked until a reply operation is performed by the receiving task. Even though the receiving task may be programmed to immediately invoke the reply primitive, it cannot perform this operation until it is selected to run from the ready to run queue. Thus the sending task remains blocked if there are a large number of ready to run tasks. If tasks that initially service event occurrences (for example the MSG and SIGNAL tasks) are kept blocked in this manner, then a greater number of events can be expected to be missed. To avoid this situation, these application tasks must be structured so that they do not themselves perform any blocking send operations. This requires the introduction of transfer tasks which in turn add to the overall system complexity. (An alternate solution is to queue the event occurrences).

The choice of whether to use a synchronous or asynchronous kernel must balance system responsiveness against application factors such as ease of understanding, requirements for transfer tasks, and simplicity of data transfer structure.

### 6.4.3 Selective Receive vs. General Receive Primitives

In the above discussion on blocking vs. nonblocking IPC protocols, it was stated that a blocking protocol eliminated the need for certain client tasks to do their own message buffer management in “handshaking” type situations. However, the imposition of a blocking protocol is not the only way to simplify the structure of application tasks. Another way is to introduce specialized primitives that provide the application tasks with more control over how IPC operations are to be performed. One such primitive is the selective receive primitive.

In this primitive, the receiving task can specify what type of message it is willing to accept\(^8\). This primitive was used in place of the general receive primitive in the

\(^8\)Instead of keying on message type, the selective receive primitive can also be designed to key
nonblocking kernel with the results presented in table 6.17. The structure of the CHNLS, DIAGS, and CINIT tasks using the selective receive primitive are presented in section B.2.3 while those using the standard receive are presented in section B.3.2.

In comparing the code for these tasks using the selective receive with that using the standard receive, it is clear that the tasks are much simpler in the case of the selective receive primitive. In this case, after the client tasks have sent the channel request message, they do not need to sift through all incoming messages waiting for the acknowledgement message. They simply call the selective receive primitive informing the kernel that they will accept only the channel acknowledgement message and that all other messages are to be queued. While this does simplify the application, it does raise the complexity of the kernel.

The results of table 6.17 show that the selective receive kernel incurs a total OS overhead of (13% + 42% + 41%) 96 across the three processors compared to 84 for the equivalent kernel with a standard receive primitive. This slight increase can in part be attributed to the fact that the average execution time for the selective receive primitive is 115 time units compared to 109 for the standard receive. The changes in the send primitives are not significantly different between the two kernels.

Another aspect of the increased system overhead is in the larger number of context switches incurred in the selective receive kernel. When a receive operation is requested using a standard receive primitive, the calling task is never blocked if there is any type of message waiting. However, in the case of the selective receive primitive, even though there may be pending messages, the calling task is still blocked if these messages are not of the required type. As a result, the system using the selective receive primitive will suffer a greater number of context switches.

Despite the slightly greater complexity of the selective receive kernel, there is not a great deal of difference between the responsiveness of the two systems. The standard receive system was able to respond to 32% of the generated events while the selective receive system was able to respond to 29%. This is largely due to the fact that in the telephone switching example, only a few tasks made use of the

---

on message source.
selective receive capabilities. In most cases, the selective receive was used just as a standard receive. As shown in section B.2.2, the way the selective receive primitive has been programmed, the queue of pending messages for the calling task must be cycled through and reordered every time a receive operation is requested on a specific message type. If no specific type is specified, then there is no need to do any message queue reordering. Thus, the cost of the selective receive primitive rises as the number of calls with specific types increases and as the average sizes of the incoming message queues grow larger. An alternative implementation of this type of primitive might be to have a separate queue for each message type. This would result in an increase in memory requirements but also a decrease in time requirements. In the example given, these two factors remained quite small so the increased cost of the selective receive was made up for by the added application simplicity.

The only way to determine the costs of such specialized primitives for a particular application is to simulate their use with a prototype and observe the system behavior as demonstrated here. After the choices regarding IPC functionality have been decided for a particular system, the only major remaining factor is task scheduling.

### 6.5 Scheduling Analysis

As discussed in section 4.1.4, there are many factors that must be considered in choosing a scheduling scheme that is appropriate for a particular application. Eventually, these come down to a number of main scheduling decisions. One of these is the determination of whether static or dynamic scheduling is warranted. In the case of static scheduling, the designer must decide on how to assign priorities and whether to use preemption or time slicing. For dynamic scheduling, the appropriate scheduling algorithm must be chosen. Finally, it must be determined whether global scheduling is necessary. In order to perform this type of scheduling analysis through prototyping, it is necessary to set up a fixed environment in terms of software partitioning, hardware platform, and IPC scheme so that any variations in prototyping results can be attributed to the characteristics of the scheduling scheme being modelled. In the
following scheduling analysis, all of the examples use the telephone switching tasks and hardware platform used in the IPC analysis section (three processors with the task to processor allocation defined in table 6.13). The IPC scheme used in this analysis is the nonblocking send with a selective blocking receive.

Again, in attempting to define a methodology for the analysis of scheduling requirements, it is difficult to provide a complete, generally applicable step by step process. To provide an optimum scheduling strategy, a priori knowledge must exist about response deadlines, computation times, task-response relationships, and event arrival times. This type of scheduling is feasible only if the system environment is static in the sense that the execution time of each response is fixed, the deadline requirement is fixed, and the number of responses to be executed is constant. However it will not work well in dynamic environments in which the processors' work loads may change over time. Although a prototyping tool is very useful in defining these parameters for a particular application, in most embedded systems, complete system knowledge remains uncertain at best and so it is not practical to construct optimal schedules for optimum execution in real-time [WC87] [LNL87] [Lac90].

Keeping in mind that the objective of the scheduling analysis is not to come up with an optimal solution but rather simply an acceptable one, it is now possible to proceed with defining the process. As in the previous analyses, there must be some yardsticks to measure the success or failure of any particular scheduling decisions. As such, the first step in the scheduling analysis process is to define a baseline system that can be compared with subsequent prototypes.

In the telephone switching example, a reasonable baseline scheduler would be one that uses static task priorities (all tasks at equal priority) with no time slicing. This is basically a first come first served (fair) scheduling scheme with no inherent bias. The MDA code used to implement this system is presented in section B.4.1. Simulation using this scheduling prototype yields the results presented in table 6.19.

Many of the fields in this table are similar to those used in the previous analyses. However, there are a few additions. The scheduling overhead field is used to determine the amount of CPU time used to perform the scheduling functions. As can be seen by
<table>
<thead>
<tr>
<th>Scheduling Method</th>
<th>Node Name</th>
<th>Final CPU Time</th>
<th>CPU Usage %</th>
<th>OS Ovhd %</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Time</td>
<td>node1</td>
<td>36857</td>
<td>57</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>node2</td>
<td>30279</td>
<td>74</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>node3</td>
<td>37527</td>
<td>93</td>
<td>42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Sched. Ovhd %</th>
<th>No. of Context Switches</th>
<th>No. of Task Preempts</th>
<th>Avge. RTR Queue Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>1</td>
<td>36</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>node2</td>
<td>3</td>
<td>91</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>node3</td>
<td>2</td>
<td>93</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Task Name</th>
<th>Task Priority</th>
<th>Total Ready Time</th>
<th>Msg Queue Size (avg,max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>msg</td>
<td>10</td>
<td>276</td>
<td>(0,1)</td>
</tr>
<tr>
<td>node2</td>
<td>synch</td>
<td>10</td>
<td>5382</td>
<td>(1,6)</td>
</tr>
<tr>
<td></td>
<td>fault</td>
<td>10</td>
<td>3578</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>audits</td>
<td>10</td>
<td>2427</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>t1m1tc</td>
<td>10</td>
<td>2827</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>diag</td>
<td>10</td>
<td>1719</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>t1diag</td>
<td>10</td>
<td>556</td>
<td>(0,0)</td>
</tr>
<tr>
<td></td>
<td>magdiag</td>
<td>10</td>
<td>490</td>
<td>(0,0)</td>
</tr>
<tr>
<td></td>
<td>toneding</td>
<td>10</td>
<td>730</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>chnldiag</td>
<td>10</td>
<td>424</td>
<td>(0,0)</td>
</tr>
<tr>
<td>node3</td>
<td>tones</td>
<td>10</td>
<td>4550</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>cinit</td>
<td>10</td>
<td>5965</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>chnls</td>
<td>10</td>
<td>8834</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>stat</td>
<td>10</td>
<td>13112</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>end</td>
<td>10</td>
<td>905</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>connect</td>
<td>10</td>
<td>4180</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>signal</td>
<td>10</td>
<td>5963</td>
<td>(0,1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event Type (# Generated) (% Completed)</th>
<th>Response Type</th>
<th>Average Completion Time</th>
<th>Overall Response Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Synch (7) (28) Synch Loss (18) (28) Msg In (23) (52) Signal Event (10) (70) Audit (6) (63)</td>
<td>Fault Report Call Setup Conn. Tone Call Takedown Audit Pass Synch Changed Diag Pass</td>
<td>12004 3988 2488 941 1380 13087 5672</td>
<td>48%</td>
</tr>
</tbody>
</table>

Table 6.19: Baseline Scheduling Scheme Results
these results, in this very straightforward static scheduler, the maximum scheduling overhead is only 3%. The preemptions field counts the number of times a task was preempted in favor of a higher priority task. Because all of the tasks in this baseline system have the same priority, the only time a preemption occurs is when the low priority IDLE task is preempted by an application task. The IDLE task runs only when there are no other ready to run tasks. The CPU usage field determines the load on each processor. This number gives the percentage of time that the processor was executing tasks other than the IDLE task. The task priority field gives the initial priority of each application task. In the baseline system, all tasks have the same priority (10). The final new field is the average completion time. This field is perhaps the most important as it gives the average amount of virtual time needed to complete a particular response using the modelled scheduler. In a complete analysis (especially for hard real time systems), the maximum time per response could also be useful. However, it has been omitted in this comparative analysis. Now that a baseline has been established, it is possible to see if the response completion times can be improved by using different scheduling methods.

6.5.1 Static Scheduling

The initial step in the scheduling process is to assume that dynamic scheduling is not necessary. Whether or not this assumption is correct for the system under consideration can be determined only by prototyping static systems and observing the results. In order to give static scheduling a fair chance, it is first necessary to come up with a priority assignment that gives improved system responsiveness.

6.5.1.1 Assigning Task Priorities

The factors involved in assigning proper task priorities have been discussed in section 4.1.4. Putting these factors into practice, one might come up with the priority assignment shown in table 6.20. The priority assigned to the MSG task is consequent as it is the only task on node1 (except for the low priority IDLE task) and will always be executed as soon as it is ready to run. However this is not the case on
<table>
<thead>
<tr>
<th>Scheduling Method</th>
<th>Node Name</th>
<th>Final CPU Time</th>
<th>CPU Usage %</th>
<th>OS Ovhd %</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC TASK No Time Slicing</td>
<td>node1</td>
<td>34513</td>
<td>75</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>node2</td>
<td>30518</td>
<td>71</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>node3</td>
<td>34774</td>
<td>98</td>
<td>47</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Sched. Ovhd %</th>
<th>No. of Context Switches</th>
<th>No. of Task Preempts</th>
<th>Avge. RTR Queue Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>1</td>
<td>41</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>node2</td>
<td>4</td>
<td>113</td>
<td>43</td>
<td>2</td>
</tr>
<tr>
<td>node3</td>
<td>3.5</td>
<td>105</td>
<td>38</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Task Name</th>
<th>Task Priority</th>
<th>Total Ready Time</th>
<th>Mag Queue Size (avg,max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>msg</td>
<td>10</td>
<td>304</td>
<td>(0,0)</td>
</tr>
<tr>
<td>node2</td>
<td>synch</td>
<td>20</td>
<td>9984</td>
<td>(1,6)</td>
</tr>
<tr>
<td></td>
<td>fault</td>
<td>30</td>
<td>304</td>
<td>(1,6)</td>
</tr>
<tr>
<td></td>
<td>audits</td>
<td>10</td>
<td>6181</td>
<td>(1,6)</td>
</tr>
<tr>
<td></td>
<td>tlmct</td>
<td>20</td>
<td>980</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>diag</td>
<td>10</td>
<td>3402</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>tldiag</td>
<td>10</td>
<td>1657</td>
<td>(0,0)</td>
</tr>
<tr>
<td></td>
<td>magdiag</td>
<td>10</td>
<td>1543</td>
<td>(0,0)</td>
</tr>
<tr>
<td></td>
<td>tonedig</td>
<td>10</td>
<td>3031</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>chnldiag</td>
<td>10</td>
<td>1749</td>
<td>(0,0)</td>
</tr>
<tr>
<td>node3</td>
<td>tones</td>
<td>30</td>
<td>284</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>cinit</td>
<td>20</td>
<td>13277</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>chnls</td>
<td>30</td>
<td>564</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>stat</td>
<td>20</td>
<td>3656</td>
<td>(1,4)</td>
</tr>
<tr>
<td></td>
<td>end</td>
<td>10</td>
<td>1218</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>connect</td>
<td>30</td>
<td>760</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>signal</td>
<td>10</td>
<td>20057</td>
<td>(0,1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event Type (# Generated) (% Completed)</th>
<th>Response Type</th>
<th>Average Completion Time</th>
<th>Overall Response Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Synch (7) (43)</td>
<td>Fault Report</td>
<td>7048</td>
<td>45%</td>
</tr>
<tr>
<td>Synch Loss (18) (33)</td>
<td>Call Setup</td>
<td>3009</td>
<td></td>
</tr>
<tr>
<td>Msg In (21) (57)</td>
<td>Conn. Tone</td>
<td>3159</td>
<td></td>
</tr>
<tr>
<td>Signal Event (11) (36)</td>
<td>Call Takedown</td>
<td>308</td>
<td></td>
</tr>
<tr>
<td>Audit (5) (60)</td>
<td>Audit Pass</td>
<td>1671</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Synch Changed</td>
<td>18247</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Diag Pass</td>
<td>695?</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.20: Static Task Priority Scheme 1 Results
the other processors.

On node2, the process starts by assigning each task a priority of 10 and proceeds by assigning more urgent tasks, higher priorities. In the switching system example, diagnostics and audits serve to look for hardware faults in the system and are of secondary importance to those tasks that are activated when a fault actually has occurred (the SYNCH, FAULT, and T1MTC tasks). These tasks are more important than the standard diagnostics as they function when a fault has actually occurred rather than simply in anticipation of a fault. For this reason, these three tasks are given an initial higher priority (20). Furthermore, the FAULT task is used by both the diagnostic subsystem and by the call processing subsystem. To prevent it from becoming a potential bottleneck, the FAULT task is assigned the highest priority (30) on node2.

The tasks on node3 are analyzed in a similar fashion. Initially, they are all assigned a priority of 10. In this set of tasks, the END and SIGNAL tasks are the least important as they are concerned with taking down calls and sampling the quality of the connections. As such, these two tasks remain at low priority. The tasks CINIT and STAT are involved in setting up new calls and thus are deemed to be more important and merit a higher priority (20). Finally, the tasks TONES, CHNLS, and CONNECT are all server tasks that are used by both the diagnostic and call processing subsystems. To prevent bottlenecks at these tasks, they are assigned the highest priority (30).

The simulation results of this prioritized static scheduling scheme using task preemption with no time slicing are presented in table 6.20. These results can now be compared with the baseline results. Use of the prioritized scheme has had an adverse effect of the total operating system overhead as it has gone up by approximately 5% on each processor. However, there has been only a marginal increase in the amount of CPU time spent on scheduling functions. The main reason for the increased system overhead is the increase in the number of preemptions and context switches due to the differing task priorities. In the baseline system, because all tasks had the same priority, there was no preemption and all were allowed to run until they were blocked.
Now that tasks have differing priorities, when a higher priority task becomes ready to run, the currently executing task is preempted and returned to the ready to run queue and a context switch is made to the newly ready task.

In examining the average response completion times, it seems that some of the times have improved while others have been degraded. The time required to report a synchronization loss has been greatly reduced from 12004 down to 7048 time units. Similarly, the time required to set up a call has declined from 3988 to 3009 time units. These improvements are directly attributable to increasing the priorities of the SYNCH, FAULT, CINIT, and STAT tasks. An added benefit of increasing the STAT task priority is that the time required to take down a call has also been reduced from 941 to 308. This is despite the fact that the END task has low priority. The reason for the improvement is that the STAT task is involved in call takedown as well as call setup.

However these improvements do not come without a cost as the remaining responses show worse completion times in the prioritized scheme. The "change synch source" response time has increased from 13087 to 18247. This is largely a consequence of the MSG task being flooded with notifications from the high priority FAULT task and thereby being unable to respond quickly to the synch change requests from the middle priority SYNCH task. Also, the diagnostic and audit responses have been degraded by approximately 1300 and 400 time units respectively. This situation was to be expected as a willing compromise was made in the priority assignment to sacrifice diagnostics for call processing.

The final degraded response is the tones connect response which now takes 670 time units longer to complete. This is curious as the TONES and CONNECT tasks are involved in the response and these are the highest priority tasks on node3. A possible explanation is that because of the relatively high priority of the CINIT task, there are many call setup requests and in turn many connect requests. When a tone connect request is sent to the CONNECT task, it is queued in a first come first served manner behind the call setup requests. Thus even though the TONES task has higher priority than the CINIT task, there is no prioritization in the message queues and
the tones request must wait. If this situation becomes a problem, the designer should consider prioritizing message queues so that high priority requests are handled first.

This initial task prioritization has sacrificed several responses in order to reduce the time needed to set up a call. During each stage of the process, it is up to the designer to decide whether such compromises are acceptable for the particular application. In the given example, the designer may want to reduce the time required for synch operations such as reporting synch loss and changing synch source. In addition, the results of table 6.20 show that the CINIT task is spending a relatively large amount of time in the ready to run state because of the higher priority server tasks. Since the CHNLS and CONNECT server tasks are used by CINIT, they run only after the CINIT task has requested their services. As such, their priorities can be set lower than that of the CINIT task. (Although the diagnostic subsystem also uses the CHNLS and CONNECT tasks, it is still considered secondary to call processing functions).

Thus the new prioritization scheme raises the priorities of the SYNCH and CINIT tasks to 30 and lowers the priorities of the CHNLS and CONNECT tasks to 20. The results of this priority assignment are shown in table 6.21. This prioritization has had a significant effect on the operating system overheads of node2. The results show that the total system overhead has been reduced from 50% to 43% and the scheduling overhead is down from 4% to 2%. The main reason for these declines is the reduction in the number of preemptions and subsequent context switches. In node2, the heaviest CPU users are the FAULT and SYNCH tasks. Now that they are at the same priority, they are unable to preempt each other thereby reducing the overheads associated with preemption. A similar but less pronounced effect is also observed in node3.

Raising the priority of the SYNCH task has drastically reduced the time it spends in the ready to run state from 9984 to 718 time units. Consequently, the times required for the synch related responses has been greatly reduced (synch loss down from 7048 to 5943 and change synch source down from 18247 to 2291). The cost of this improvement has been the increase of almost all other response times. In most cases,
<table>
<thead>
<tr>
<th>Scheduling Method</th>
<th>Node Name</th>
<th>Final CPU Time</th>
<th>CPU Usage %</th>
<th>OS Ovhd %</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC TASK. No Time Slicing</td>
<td>node1</td>
<td>37434</td>
<td>82</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>node2</td>
<td>32348</td>
<td>69</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>node3</td>
<td>36949</td>
<td>97</td>
<td>46</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Sched. Ovhd %</th>
<th>No. of Context Switches</th>
<th>No. of Task Preempts</th>
<th>Avge. RTR Queue Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>1</td>
<td>24</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>node2</td>
<td>2</td>
<td>79</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>node3</td>
<td>3</td>
<td>97</td>
<td>29</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Task Name</th>
<th>Task Priority</th>
<th>Total Ready Time</th>
<th>Msg Queue Size (avg,max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>msg</td>
<td>10</td>
<td>192</td>
<td>(0,9)</td>
</tr>
<tr>
<td>node2</td>
<td>synch</td>
<td>30</td>
<td>718</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>fault</td>
<td>30</td>
<td>421</td>
<td>(1,3)</td>
</tr>
<tr>
<td></td>
<td>audits</td>
<td>10</td>
<td>609</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>timt</td>
<td>20</td>
<td>310</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>diag</td>
<td>10</td>
<td>6941</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>t1diag</td>
<td>10</td>
<td>6312</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>msgdiag</td>
<td>10</td>
<td>1813</td>
<td>(0,0)</td>
</tr>
<tr>
<td></td>
<td>toning</td>
<td>10</td>
<td>2454</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>chnldiag</td>
<td>10</td>
<td>1699</td>
<td>(0,0)</td>
</tr>
<tr>
<td>node3</td>
<td>tones</td>
<td>30</td>
<td>293</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>cinit</td>
<td>30</td>
<td>303</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>chns</td>
<td>20</td>
<td>7253</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>stat</td>
<td>20</td>
<td>9014</td>
<td>(1,3)</td>
</tr>
<tr>
<td></td>
<td>end</td>
<td>10</td>
<td>854</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>connect</td>
<td>20</td>
<td>7760</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>signal</td>
<td>10</td>
<td>22945</td>
<td>(0,1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event Type (# Generated) (% Completed)</th>
<th>Response Type</th>
<th>Average Completion Time</th>
<th>Overall Response Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Synch (5) (40) Synch Loss (11) (45) Msg In (25) (58) Signal Event (8) (67) Audit (4) (0)</td>
<td>Fault Report</td>
<td>5843</td>
<td>49%</td>
</tr>
<tr>
<td></td>
<td>Call Setup</td>
<td>4854</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Conn. Tone</td>
<td>4101</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Call Takedown</td>
<td>298</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Integ Fail Diag</td>
<td>12255</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Synch Changed</td>
<td>2261</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Diag Pass</td>
<td>7741</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.21: Static Task Priority Scheme 2 Results
these increases are relatively small. However, the time required for the call setup response has increased by 1845 time units. This sizeable increase of approximately 60% may be due to the fact that the MSG task is now being flooded with many more synch related requests and thus requires more time to initiate the call setup responses. Again, it is up to the designer and end user to decide whether this is an acceptable compromise.

6.5.1.2 Assigning Response Priorities

All of the previous priority assignment methods have assumed that because tasks are the schedulable entities in the system, it is the tasks that should be assigned the individual priorities. However, the execution of an individual task provides only a partial response. As the response makes its way through its various constituent tasks, its relative priority changes depending on what stage it has reached in its completion. This does not seem to be a reasonable approach as the importance of providing a response should not be based on the nature of the tasks that make up the response. In fact, the importance of the tasks should be based on the type of response that they execute.

One way to achieve this type of response scheduling might be to assign a priority to each response and then assign this same priority to every task that is a part of that response. For tasks that are a part of more than one response, the associated responses’ priorities should be combined to determine the final priority of the task in question. While this type of approach does allow response type to dictate task priorities, it still places fixed priority constraints on individual tasks. For instance, consider the CONNECT task. This task may be a part of a diagnostic response and it is also a part of the call setup response. As such, it would rate a fairly high priority. This high priority is justified for the call setup response but not for the diagnostic response. Because of its fixed high priority, the CONNECT task would always be given the processor over call processing tasks such as CINIT, STAT, and END even if it were servicing a diagnostic request.

A better way to implement response scheduling is to still associate a priority
with each type of response but to not use these response priorities to establish fixed
task priorities. Instead, the tasks themselves should not have any fixed priority; they
should take on the priority of the response they are servicing at the particular instance.
Thus in the case of the CONNECT task, when it is servicing a diagnostic response,
it will take on the low priority of the diagnostic response. When it is servicing a call
processing response, it will take on the high priority of the call processing response.

The practical implementation of this response scheduling scheme is given in sec-
tions B.4.3 and B.4.4. The priorities of the responses of the telephone switching
example have been assigned as follows:

1. Setup new call (90).
2. Connect tone (80).
4. Change synch source (70).
5. Takedown existing call (70).
6. Complete diagnostic (no failure) (60).
7. Complete audit (no failure) (60).

In order to be able to propagate a response priority from task to task, the priority
of the response is added as a tag to every message in the system. Thus whenever
an event is generated, the kernel makes the initial responding task ready to run with
the priority associated with the response to be generated. If it is a complex response
(requires more than one task), then each task in the response tags each message with
the response priority.

In the kernel shown in section B.4.3, the send and receive primitives have been
slightly modified to allow message passed response priorities. If a task is waiting for
a message, when the message is actually received, its priority tag is examined and the
receiving task is given that priority and placed on the ready to run queue. When it
finishes servicing the response, it again becomes blocked waiting for a message. The
next time around, this task may have a different priority depending on the priority of the received message. In addition to the kernel, the application tasks have been changed to apropriately fill in the priority tag of each outgoing message.

This priority propagation via message tags has an additional benefit in that it allows the priority of a response to change to reflect changing circumstances. Consider the case of a diagnostic request. Initially, the associated response will have a relatively low priority of 60. If there is no failure detected, the response will be propagated through the system with a priority of 60. However, if a failure is detected, the message sent to the FAULT task can be sent with a tag indicating a higher priority of 80. From this point, the response has thus acquired a higher priority and will be sent through the system with a greater urgency.

A prototype of this type of priority assignment was used to obtain the results presented in table 6.22. In this simulation, all tasks were given an initial priority of 10 and then their priorities were allowed to vary as dictated by the responses that they were executing. In terms of operating system overheads, the results are very similar to those obtained with the second task priority assignment of table 6.21. This is perhaps because the task priority assignment was also done by considering response priorities and then mapping them to task priorities. This resulted in similar preemption and context switching overheads. In any case, there is a slight improvement in the overheads associated with node3 due to fewer preemptions and context switches. This is perhaps due to the fact that since all tasks servicing a particular response have the same priority, they can not preempt one another. If several such responses occur in succession, there is even less chance of preemption as many of the active tasks will have acquired the same priority.

The assignment of priorities to responses has a very strong impact on the average completion times associated with the responses. The results are compared with the last task priority assignment of table 6.21 as this assignment has given the best overall results, on average, so far. The call setup response had the highest priority and its completion time has been reduced to 3115 from 4101 in the case of task priorities and from 3988 in the baseline case. Similarly, the time required to report a fault has been
<table>
<thead>
<tr>
<th>Scheduling Method</th>
<th>Node Name</th>
<th>Final CPU Time</th>
<th>CPU Usage %</th>
<th>OS Ovhd %</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC RESP. No Time</td>
<td>node1</td>
<td>36805</td>
<td>57</td>
<td>15</td>
</tr>
<tr>
<td>Slicing</td>
<td>node2</td>
<td>31596</td>
<td>63</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>node3</td>
<td>36665</td>
<td>85</td>
<td>42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Sched. Ovhd %</th>
<th>No. of Context Switches</th>
<th>No. of Task Preempts</th>
<th>Avg. RTR Queue Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>1</td>
<td>36</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>86</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>87</td>
<td>14</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Task Name</th>
<th>Task Priority</th>
<th>Total Ready Time</th>
<th>Msg Queue Size (avg,max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>msg</td>
<td>10</td>
<td>277</td>
<td>(0,4)</td>
</tr>
<tr>
<td>node2</td>
<td>synch</td>
<td>10</td>
<td>738</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>fault</td>
<td>10</td>
<td>464</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>audits</td>
<td>10</td>
<td>1739</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>timtc</td>
<td>10</td>
<td>927</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>diags</td>
<td>10</td>
<td>3645</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>tidings</td>
<td>10</td>
<td>3832</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>msgdiag</td>
<td>10</td>
<td>1737</td>
<td>(0,0)</td>
</tr>
<tr>
<td></td>
<td>tonedig</td>
<td>10</td>
<td>1714</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>chndiag</td>
<td>10</td>
<td>1677</td>
<td>(0,0)</td>
</tr>
<tr>
<td>node3</td>
<td>tones</td>
<td>10</td>
<td>1279</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>cinit</td>
<td>10</td>
<td>1526</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>chnls</td>
<td>10</td>
<td>6574</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>stat</td>
<td>10</td>
<td>12099</td>
<td>(1,3)</td>
</tr>
<tr>
<td></td>
<td>end</td>
<td>10</td>
<td>1197</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>connect</td>
<td>10</td>
<td>3315</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>signal</td>
<td>10</td>
<td>11146</td>
<td>(0,1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event Type (## Generated) (% Completed)</th>
<th>Response Type (PRIORITY)</th>
<th>Average Completion Time</th>
<th>Overall Response Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1403</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>961</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8164</td>
</tr>
</tbody>
</table>

Table 6.22: Static Response Priority Scheme Results
reduced to 3501 time units from 5943 and 12004 units. The connect tone response time has been greatly reduced from 4101 and 2408 down to 1403 time units.

As always the improvements in the high priority responses are accompanied by degradations in the low priority responses. In the case where priorities were assigned to individual tasks, those relatively unimportant responses that used some of the same tasks as high priority responses enjoyed a performance boost due to the high task priorities. This is no longer the case as each task takes on the priority of the response. Hence the boost is gone and the completion times of the change synch source, call takedown, diagnostic, and audit responses have suffered.

In this particular case, the results obtained from assigning priorities to complete responses rather than to individual tasks do not show dramatic differences although some of the differences are significant. As mentioned previously, the main reason for this is that in the analysis using task priorities, response priorities were manually mapped to the individual tasks. In larger systems, this approach may become quite cumbersome if not impossible. Assigning priorities to complete responses provides a convenient and flexible way to ensure that urgent responses are serviced before those of lesser importance.
6.5.1.3 Time Slicing

In all of the examples presented so far, the static scheduling systems have allowed tasks to run until they are logically blocked or until they are preempted by tasks of higher priority. That is, there was no time slicing. In time slicing, the available CPU time is divided into fixed length segments and a task is allowed to execute until it is blocked, preempted, or until its segment (time slice) expires. Time slicing has the effect of preventing one task from monopolizing the processor while other tasks of equal priority remain ready to run. As such, time slicing should be most effective when the processor contains many tasks of equal priority. In contrast, if most of the tasks are of different priority, it is likely that when a time slice occurs, there are no other tasks of a priority equal to that of the running task. As such, the running tasks is simply momentarily swapped from the running state to the ready to run state and then immediately back to the running state. This is obviously a needless overhead.

In the kernels used in this scheduling analysis, time slicing is achieved by adding a cyclic clock interrupt to each processor. When this interrupt is raised, the state of the currently executing task is saved and it is put back on the ready to run queue according to its priority (if there are other ready to run tasks with the same priority, the current task is placed behind these tasks). Finally, the next task is removed from the ready to run queue and it is given control of the processor. The length of the time slice (time between cyclic interrupt occurrences) can be varied but has been arbitrarily chosen to correspond to approximately 300 time units. In light of the fact that there is no need for time slicing on a processor with all tasks at different priorities or on a processor with only one task, time slicing has not been implemented in node1 in the following examples. The modifications made to the kernel to incorporate time slicing are given in section B.4.2.

In order to get an understanding of the effects of time slicing, prototypes of the static task priority system of table 6.21 and the static response priority system of table 6.22 were built using time slicing. The results obtained from their use are given in tables 6.23 and 6.24 respectively.

In comparing these results with those obtained without time slicing, it seems that
<table>
<thead>
<tr>
<th>Scheduling Method</th>
<th>Node Name</th>
<th>Final CPU Time</th>
<th>CPU Usage %</th>
<th>OS Ovhd %</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC TASK. With Time Slicing</td>
<td>node1</td>
<td>35593</td>
<td>63</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>node2</td>
<td>32719</td>
<td>51</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>node3</td>
<td>35117</td>
<td>91</td>
<td>51</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Sched. Ovhd %</th>
<th>No. of Context Switches</th>
<th>No. of Task Preempts</th>
<th>Avge. RTR Queue Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>1</td>
<td>33</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>node2</td>
<td>4</td>
<td>162</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>node3</td>
<td>5</td>
<td>165</td>
<td>26</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Task Name</th>
<th>Task Priority</th>
<th>Total Ready Time</th>
<th>Msg Queue Size (avg,max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>msg</td>
<td>10</td>
<td>249</td>
<td>(0,6)</td>
</tr>
<tr>
<td>node2</td>
<td>synch</td>
<td>30</td>
<td>3563</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>fault</td>
<td>30</td>
<td>476</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>audits</td>
<td>10</td>
<td>2393</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>t1mct</td>
<td>20</td>
<td>373</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>diags</td>
<td>10</td>
<td>4298</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>t1diag</td>
<td>10</td>
<td>2726</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>magdiag</td>
<td>10</td>
<td>1600</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>tonediag</td>
<td>10</td>
<td>1198</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>chndiag</td>
<td>10</td>
<td>1832</td>
<td>-</td>
</tr>
<tr>
<td>node3</td>
<td>tones</td>
<td>30</td>
<td>164</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>cinit</td>
<td>30</td>
<td>609</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>chns</td>
<td>20</td>
<td>9121</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>stat</td>
<td>20</td>
<td>9232</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>end</td>
<td>10</td>
<td>1977</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>connect</td>
<td>20</td>
<td>7007</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>signal</td>
<td>10</td>
<td>21851</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event Type (# Generated) (% Completed)</th>
<th>Response Type</th>
<th>Average Completion Time</th>
<th>Overall Response Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Synch (6) (17)</td>
<td>Fault Report</td>
<td>7150</td>
<td>46%</td>
</tr>
<tr>
<td>Synch Loss (10) (60)</td>
<td>Call Setup</td>
<td>2364</td>
<td></td>
</tr>
<tr>
<td>Msg In (19) (58)</td>
<td>Conn. Tone</td>
<td>2116</td>
<td></td>
</tr>
<tr>
<td>Signal Event (11) (36)</td>
<td>Call Takedown</td>
<td>446</td>
<td></td>
</tr>
<tr>
<td>Audit (4) (25)</td>
<td>Audit</td>
<td>558</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Synch Changed</td>
<td>2514</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Diag Pass</td>
<td>9047</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.23: Static Task Priorities With Time Slicing
<table>
<thead>
<tr>
<th>Scheduling Method</th>
<th>Node Name</th>
<th>Final CPU Time</th>
<th>CPU Usage %</th>
<th>OS Ovhd %</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC RESP. With Time Slicing</td>
<td>node1</td>
<td>34100</td>
<td>53</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>node2</td>
<td>28207</td>
<td>76</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>node3</td>
<td>37271</td>
<td>75</td>
<td>48</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Sched. Ovhd %</th>
<th>No. of Context Switches</th>
<th>No. of Task Preempts</th>
<th>Avge. RTR Queue Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>2</td>
<td>49</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>node2</td>
<td>7</td>
<td>191</td>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td>node3</td>
<td>5</td>
<td>171</td>
<td>13</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Task Name</th>
<th>Task Priority</th>
<th>Total Ready Time</th>
<th>Mag Queue Size (avg,max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>msg</td>
<td>10</td>
<td>361</td>
<td>(0,6)</td>
</tr>
<tr>
<td>node2</td>
<td>synch</td>
<td>10</td>
<td>5717</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>fault</td>
<td>10</td>
<td>2019</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>audits</td>
<td>10</td>
<td>3045</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>tlmc</td>
<td>10</td>
<td>2755</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>diags</td>
<td>10</td>
<td>3798</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>t1diag</td>
<td>10</td>
<td>1426</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>msgdiag</td>
<td>10</td>
<td>585</td>
<td>(0,0)</td>
</tr>
<tr>
<td></td>
<td>tone</td>
<td>10</td>
<td>1938</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>chnlng</td>
<td>10</td>
<td>1010</td>
<td>(0,0)</td>
</tr>
<tr>
<td>node3</td>
<td>tones</td>
<td>10</td>
<td>2249</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>cinit</td>
<td>10</td>
<td>5405</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>chnl</td>
<td>10</td>
<td>6417</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>stat</td>
<td>10</td>
<td>4976</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>end</td>
<td>10</td>
<td>357</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>connect</td>
<td>10</td>
<td>4218</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>signal</td>
<td>10</td>
<td>2458</td>
<td>(0,1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event Type (# Generated) (% Completed)</th>
<th>Response Type (PRIORITY)</th>
<th>Average Completion Time</th>
<th>Overall Response Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Synch (6) (20) Synch Loss (22) (55) Msg In (18) (67) Signal Event (5) (60) Audit (4) (25)</td>
<td>Fault Report (60) Call Setup (90) Conn. Tone (80) Call Takedown (70) Audit Pass (60) Synch Changed (70) Diag Pass (60)</td>
<td>5472 6253 4553 548 1337 6088 4924</td>
<td>51%</td>
</tr>
</tbody>
</table>

Table 6.24: Static Response Priorities With Time Slicing
time slicing causes a small increase in the amount of operating system overhead (about 3% per processor). This increase is also reflected to the same degree in the scheduling overhead. The number of context switches required in both of the systems increased dramatically with the introduction of time slicing. The response completion data must now be analyzed to determine whether the extra overheads have been compensated for.

If time slicing has been effective, it should be evident in the response completion times for those responses that use tasks of the same priority (in the case of task priorities) or in the completion times for responses of the same priority (in the case of response priorities). In both cases, the response times of shorter responses should be improved somewhat while the times of the longer responses should be degraded somewhat. This is because the short responses are no longer completely shut out until a particular task completes its portion of a long response.

The results of the telephone switching examples do not bear out this conjecture. In considering the task priority example of table 6.23, all of the diagnostic tasks are of the same priority but they do not benefit from time slicing as they do not run with enough frequency so that two or more of them are ready at the same time. The results show a slight degradation in the diagnostic responses requested by the switching center. The results also show a degradation in the fault report, synch source change, and call takedown responses. These degradations are likely due to the increased context switching and operating system overhead of the time slicing scheme. In contrast, there is significant improvement in the call setup and tone connect responses. These improvements may be due to more equitable processor sharing on node3 between these two response types.

The results obtained from the time slicing version of the response priority system of table 6.24 are no more conclusive. The results show improvements in the change synch source, call takedown, and diagnostic responses and degradations in the fault report, tone connect, call setup, and audit responses. In most cases, the improvements in response completion times are overshadowed by the degradations of the remaining response completion times.
The only consistent factor in the time slicing analysis for the telephony example is that slicing increases system overheads. In this case, there were no real benefits in including time slicing. This is likely due to the nature of the application. In the example system, there are not very many responses or tasks with the same priorities. As such, it is a relatively rare occasion when two tasks of the same priority are ready to run on the same processor at the same time (the required conditions for effective time slicing). Also, it may be that the task cycle times are not long enough for time slicing to have a significant effect. In this case, an executing task would be able to complete before its time slice expired. It is possible to make the time slices smaller in this case but this simply increases system overhead. If the task cycle times are so small, it may be better to allow a task to complete as the ready tasks are only blocked for small periods.

Another reason that time slicing seems to be ineffective in this example is that the constituent activities of the tasks are modelled as fixed time delays through the use of the MDA ticks instruction. Execution of this instruction simply causes an increase in elapsed processor time. The actual execution of the instruction is indivisible and so a time slice can not occur during the elapsed time. A true representation of the required activity time can be acheived by modelling the activities as tight loops executing a "ticks #1" instruction for the required number of iterations. In this way, a check is continually made for the occurrence of the time slice interrupt during the execution of the activity.

Despite this shortcoming in the model, it seems that the telephone switching system is not suitable for the inclusion of time slicing. However, this fact was not apparent until the system was modelled using the discussed prototype.

This analysis has provided an idea as to the type of responsiveness that can be expected from the system employing purely static scheduling techniques. The next step in the process is to determine whether or not this responsiveness can be improved by using a dynamic scheduling scheme.
6.5.2 Dynamic Scheduling

There are many different types of dynamic scheduling algorithms used in real time systems. The purpose of this thesis is not to expound on the details into the workings and the complexities of any one of these. The details of the scheduling techniques covered in this thesis can be found in [Cra88] and [Lac90]. Rather, the focus here is on demonstrating the prototyping method that can be used to model and evaluate the effectiveness of the different techniques for a specific application.

The main reason for considering the use of dynamic scheduling algorithms is that real time systems operate in a dynamic environment and must adapt to a changing workload. Instead of making decisions solely on the basis of the current state of the system, previous history and expected arrivals must also be taken into account [WC87]. However, this adaptability does not come for free, the run-time cost of the chosen scheduling algorithm must also be considered.

There are many dynamic scheduling algorithms including shortest task first, longest task first, earliest response due date, and least response slack time. Building prototypes using each of these (and perhaps others) is used to determine whether the benefits of any of these schemes offset the overheads associated with them. Each of these four methods is now examined for the telephone switching example.

6.5.2.1 Shortest Task First

The “shortest task first” (STF) scheduling algorithm simply uses the expected run time of the ready to run tasks to determine which will be the next to run. The task that has the shortest expected execution time will be put at the front of the ready to run queue. One way to implement this algorithm (as done in the prototype presented in section B.4.5) is to simply assign static task priorities based on the predetermined task execution times with the tasks having shorter cycle times being assigned higher priorities.

The major aspect of this scheduling scheme is determining the execution times of the individual tasks. One approach is to always identify the maximum processor requirements as determined by the worst case program flow through the task [SR87].
While this is relatively easy to do, it may not give the best approximation as to the
time each task will ordinarily require. A better way to determine execution time is
to determine each task's average cycle time.

A task's cycle time is defined as being the amount of processor time that it requires
to perform its part of a response. Because individual tasks can be part of more than
one response and because the same partial response can take different amounts of time
at each iteration, an average measure is more useful than the maximum measure. In
the prototype used to obtain the results presented in table 6.25, the average cycle
times of tasks were determined by running simulations using a first-come first-served
scheduling scheme (all tasks at equal priority) and having each task determine its
average cycle time by recording its execution time and dividing by the number of
cycles it had done. These cycle times are given in table 6.25. From these cycle times,
static task priorities were assigned. Because each task had a different average cycle
time, they all warranted different priorities. As such, there was no need for time
slicing.

While this is not truly a dynamic scheduling approach, it does provide a good
approximation. A truer dynamic shortest task first algorithm would continuously
calculate and update the average task cycle times as the system progressed. In this
way, previous system behavior would have a direct effect on future scheduling deci-
sions. However, because the tasks comprising the example system do not show wide
variations in execution times between iterations, the only shortcoming of the mod-
elled prototype is that it may not provide a true indication of the overhead involved
in dynamically updating the average task execution times. However, the response
statistics should still be applicable.

In examining table 6.25, it is apparent that even though not all aspects of the
dynamic scheduling algorithm have been implemented in the prototype, there is an
increase in the amount of operating system overhead. (The lack of increase in schedul-
ing overhead is a testimony to the fact that average task cycle times are not dynami-
cally calculated and updated). The increase in the total overhead is likely due to the
fact that the system requires relatively large numbers of context switches and task

241
<table>
<thead>
<tr>
<th>Scheduling Method</th>
<th>Node Name</th>
<th>Final CPU Time</th>
<th>CPU Usage %</th>
<th>OS Ovhd %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shortest Task</td>
<td>node1</td>
<td>31118</td>
<td>72</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>node2</td>
<td>26929</td>
<td>75</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>node3</td>
<td>30102</td>
<td>81</td>
<td>47</td>
</tr>
<tr>
<td>First</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Sched Ovhd %</th>
<th>No. of Context Switches</th>
<th>No. of Task Preempts</th>
<th>Avge. RTR Queue Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>1</td>
<td>38</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>node2</td>
<td>3</td>
<td>113</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>node3</td>
<td>3</td>
<td>96</td>
<td>28</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Task Name</th>
<th>Avg Cycle Time</th>
<th>Total Ready Time</th>
<th>Msg Queue Size (avg,max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>msg</td>
<td>552</td>
<td>291</td>
<td>(0,4)</td>
</tr>
<tr>
<td>node2</td>
<td>synch</td>
<td>100</td>
<td>348</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>fault</td>
<td>325</td>
<td>5286</td>
<td>(0,3)</td>
</tr>
<tr>
<td></td>
<td>audits</td>
<td>243</td>
<td>417</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>tlimct</td>
<td>258</td>
<td>316</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>diag</td>
<td>275</td>
<td>2812</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>ttdiag</td>
<td>496</td>
<td>2077</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>msgdiag</td>
<td>287</td>
<td>1234</td>
<td>(0,0)</td>
</tr>
<tr>
<td></td>
<td>tnediag</td>
<td>401</td>
<td>3077</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>chndiag</td>
<td>234</td>
<td>370</td>
<td>(0,1)</td>
</tr>
<tr>
<td>node3</td>
<td>tones</td>
<td>247</td>
<td>605</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>cinit</td>
<td>178</td>
<td>1848</td>
<td>(1,3)</td>
</tr>
<tr>
<td></td>
<td>chnl</td>
<td>56</td>
<td>341</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>stat</td>
<td>230</td>
<td>2054</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>end</td>
<td>184</td>
<td>282</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>connect</td>
<td>503</td>
<td>9468</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>signal</td>
<td>283</td>
<td>7647</td>
<td>(0,1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event Type (# Generated)</th>
<th>Response Type</th>
<th>Average Completion Time</th>
<th>Overall Response Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Synch (5) (20)</td>
<td>Fault Report</td>
<td>2439</td>
<td>52%</td>
</tr>
<tr>
<td>Synch Loss (16) (56)</td>
<td>Call Setup</td>
<td>4204</td>
<td></td>
</tr>
<tr>
<td>Msg In (16) (69)</td>
<td>Conn. Tone</td>
<td>3172</td>
<td></td>
</tr>
<tr>
<td>Signal Event (5) (40)</td>
<td>Call Takedown</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>Audit (4) (25)</td>
<td>Diag Failure</td>
<td>16677</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Synch Changed</td>
<td>2502</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Diag Pass</td>
<td>7268</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.25: Shortest Task First System Results
preemptions when compared to the baseline system. This is especially evident in
node2 with a total overhead of 59% due to 113 context switches and 36 task preem-
ptions. These types of statistics are to be expected in the shortest task first scheme.
Because the system favors those tasks with shorter execution times, it is expected that
a larger number of different tasks will need to be prepared for execution in a shorter
time period. The total overhead could be reduced by eliminating task preemptions
and allowing the tasks to complete. This is, in fact, the most common manner of
implementation.

In determining the effects of the STF scheduling scheme on response completions,
several trends become apparent. The first of these is that this scheme allows the
capture and completion of a relatively large percentage of event occurrences (52% in
this case). This can be justified by observing another trend. Because the algorithm
favors responses composed of short execution segments, the response completion times
of many of the shorter responses such as synch loss and synch source change have
been greatly reduced from those times produced by the baseline system. As such, the
tasks servicing these types of events spend less time on the ready to run queue and
more time capturing event occurrences. This “pushing through” of short responses
accounts for the increase in the response completion percentage.

However, because the total amount of processing capacity remains the same in
all cases, the increased throughput of responses with shorter components is offset by
an increase in the completion times of those responses that are composed of longer
execution segments. Three of the more notable cases in this example are the call
setup, tone connection, and diagnostic responses. All of these responses require the
use of the CONNECT task which has a long execution time of 503 time units. As such,
it is always placed at the end of the ready to run queue. Consequently, the completion
times for those responses using this and other long tasks are significantly increased
over their times in the baseline system. While the STF scheme does produce generally
higher response throughputs, it is inevitable that during prolonged or heavy periods
of system activity, the completion times of the longer responses will be degraded
to unacceptable levels. This is especially true in the example system as the longer
responses such as call setup and tone connection are quite important. To prevent this situation, the opposite approach might be tried.

6.5.2.2 Longest Task First

The "longest task first" (LTF) scheduling algorithm uses the exact same methods as the STF algorithm to determine task cycle times. Also like the STF case, the implementation presented in section B.4.6 predetermines the task cycle times through simulation rather than calculating and updating them dynamically. The only difference between the STF and LTF algorithms is that in the case of LTF, the task with the longer task cycle time is scheduled for execution ahead of the task with the shorter cycle time. The results of the prototype using this algorithm are presented in table 6.26.

In examining table 6.26, there are some marked changes over the results obtained with the STF algorithm. Just as there was an increase in the amount of system overhead with the STF scheme, there is a significant decrease in this statistic in the LTF case with the highest overhead being only 37%. The cause of this decrease is the opposite of the cause of the increase in the STF case. With the LTF algorithm, the system requires relatively fewer context switches and task preemptions when compared to the baseline system. The LTF system favors those tasks with longer execution times and so fewer different tasks will need to be prepared for execution in a given time period.

Like the incurred system overheads, some of the effects of the LTF scheduling scheme on response completions are opposite to those of the STF scheme. For example, while the STF scheme allowed the capture and completion of a very high percentage of event occurrences, the LTF scheme allows for the completion of a very low percentage of event occurrences (only 26%). The numerous short responses are no longer being pushed through and their event servicing tasks are being stranded on the ready to run queue for longer periods behind more time consuming responses. For example, the SYNCH task spends 7179 time units on the ready queue as opposed to only 348 in the STF case. As a result, these tasks are unable to capture the latched
<table>
<thead>
<tr>
<th>Scheduling Method</th>
<th>Node Name</th>
<th>Final CPU Time</th>
<th>CPU Usage %</th>
<th>OS Ovhd %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Longest Task</td>
<td>node1</td>
<td>32148</td>
<td>62</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>node2</td>
<td>35464</td>
<td>58</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>node3</td>
<td>39267</td>
<td>69</td>
<td>37</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Sched. Ovhd %</th>
<th>No. of Context Switches</th>
<th>No. of Task Preempts</th>
<th>Avge. RTR Queue Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>1</td>
<td>34</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>node2</td>
<td>2</td>
<td>78</td>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>node3</td>
<td>3</td>
<td>82</td>
<td>25</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Task Name</th>
<th>Avge. Cycle Time</th>
<th>Total Ready Time</th>
<th>Msg Queue Size (avg,max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>msg</td>
<td>552</td>
<td>263</td>
<td>(0,3)</td>
</tr>
<tr>
<td>node2</td>
<td>synch</td>
<td>100</td>
<td>7197</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>fault</td>
<td>325</td>
<td>423</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>audits</td>
<td>243</td>
<td>7922</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>t1mtc</td>
<td>258</td>
<td>1534</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>diags</td>
<td>275</td>
<td>3236</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>t1diag</td>
<td>496</td>
<td>184</td>
<td>(0,0)</td>
</tr>
<tr>
<td></td>
<td>msdgdiag</td>
<td>287</td>
<td>580</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>tonedias</td>
<td>401</td>
<td>384</td>
<td>(0,0)</td>
</tr>
<tr>
<td></td>
<td>chnldiag</td>
<td>234</td>
<td>1619</td>
<td>(0,0)</td>
</tr>
<tr>
<td>node3</td>
<td>tones</td>
<td>247</td>
<td>2681</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>cinit</td>
<td>178</td>
<td>11998</td>
<td>(0,3)</td>
</tr>
<tr>
<td></td>
<td>chnls</td>
<td>56</td>
<td>13893</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>stat</td>
<td>230</td>
<td>4016</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>end</td>
<td>184</td>
<td>655</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>connect</td>
<td>503</td>
<td>308</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>signal</td>
<td>283</td>
<td>713</td>
<td>(0,1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event Type</th>
<th>(# Generated)</th>
<th>Response Type</th>
<th>Average Completion Time</th>
<th>Overall Response Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Synch</td>
<td>(12) (0)</td>
<td>Fault Report</td>
<td>2057</td>
<td>26%</td>
</tr>
<tr>
<td>Synch Loss</td>
<td>(27) (11)</td>
<td>Call Setup</td>
<td>7596</td>
<td></td>
</tr>
<tr>
<td>Msg In</td>
<td>(30) (33)</td>
<td>Conn. Tone</td>
<td>4600</td>
<td></td>
</tr>
<tr>
<td>Signal Event</td>
<td>(8) (13)</td>
<td>Call Takedown</td>
<td>538</td>
<td></td>
</tr>
<tr>
<td>Audit</td>
<td>(8) (100)</td>
<td>Audit Pass</td>
<td>518</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Audit Fail</td>
<td>2015</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Diag Pass</td>
<td>6125</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.26: Longest Task First System Results
event occurrences.

Theoretically, the LTF scheme should favor responses composed of long execution segments. As such, it would be expected that the completion times of those responses consisting of longer execution segments would be decreased from the corresponding times using the STF algorithm. However, the results of table 6.26 show that this is not the case. In comparing these results with those of the STF results of table 6.25, it seems that short responses such as synch loss have remained relatively unchanged (even improved somewhat) while the longer responses such as tone connect and call setup have suffered degradations. There are two reasons for these observations.

The first is that while the longer responses are favored for execution, there are several such responses and while one is executing, the others must wait. This situation is really no different from the STF case except that now, instead of waiting for short tasks to finish, all tasks must wait for longer tasks to relinquish the processor. Thus there is no real benefit to the longer responses. The second reason for the degradation of the longer responses is due to the nature of the responses in the telephony example. The call setup response requires the services of the CONNECT task and also of the CHNLS task. While the CONNECT task is favored for execution due to its long cycle time, the CHNLS task is delayed due to its short execution time of only 56 time units. This disparity is reflected in the amount of time these tasks spend on the ready to run queue (13893 time units for the CHNLS task and only 308 units for the CONNECT task). Thus compound responses such as call setup which use both short and long tasks are disfavored in both the STF and LTF scheduling schemes.

Despite the shortcomings of the LTF scheduling scheme for this particular application, it has resulted in a decrease in the diagnostic response completion times. While the diagnostic responses also use the CHNLS task, their occurrence is less frequent than the call setup responses. As such, it is possible that in the course of the simulation, the diagnostic events were processed at times when they did not have to compete with other long responses.

From the examinations of the STF and LTF scheduling schemes, it seems that
they pay little attention to the perceived urgency of the individual responses and simply attempt to increase the overall response throughput of the system by increasing the percentage of captured events or by decreasing OS overheads. It was observed that the STF scheme was more successful than the LTF scheme in achieving this goal for the telephony application. If all of the responses of a particular application are of equal importance, the STF algorithm may be quite useful. However, in the switching example, the objective is to selectively increase the throughput of specific responses. The "earliest response due date" and "least response slack time" algorithms are perhaps better suited for this purpose.

6.5.2.3 Earliest Response Due Date

The "earliest (response) due date" (EDD) algorithm schedules tasks based on the time at which the response currently being processed by the task, is required. For systems with different response priorities, this scheduling algorithm is favored because it bases its decisions on the characteristics of complete responses rather than on the characteristics of individual tasks (as was the case with the STF and LTF schemes).

In the EDD algorithm implemented in the prototype presented in sections B.4.7 and B.4.8, as each event is triggered in the kernel, it is assigned a due date time. This due date is based on the time of the event occurrence, on the perceived urgency of the response associated with the event, and on the time required to complete a response to the event. In considering these factors, a response time is calculated (statically) for each event. The due date is then dynamically calculated by adding this response time to the time of the event occurrence. The response times used for the events in the telephony example are as follows.
<table>
<thead>
<tr>
<th>EVENT</th>
<th>RESPONSE TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change synch source</td>
<td>900 units</td>
</tr>
<tr>
<td>Synch loss (fault report)</td>
<td>700 units</td>
</tr>
<tr>
<td>Call setup</td>
<td>5000 units</td>
</tr>
<tr>
<td>Connect tone</td>
<td>3000 units</td>
</tr>
<tr>
<td>Diagnostic pass</td>
<td>12000 units</td>
</tr>
<tr>
<td>Handle integrity okay</td>
<td>2000 units</td>
</tr>
<tr>
<td>Handle integrity failure</td>
<td>1000 units</td>
</tr>
<tr>
<td>(fault report)</td>
<td></td>
</tr>
<tr>
<td>Call takedown</td>
<td>700 units</td>
</tr>
<tr>
<td>Perform audit</td>
<td>1100 units</td>
</tr>
</tbody>
</table>

These response times convey the relative importance of each event with due consideration to the best times (from previous results) required for their servicing. Several prototyping iterations can be performed later and the response times can be tightened at each iteration until reasonable numbers are found.

As each of these events occurs during run time, its due date is calculated and posted for the initial task handling the event. From this point, the due date for the response is propagated from task to task as a tag field in each message generated in the execution of the response. In a manner similar to that used in the static response priority scheme of section 6.5.1.2, the kernel schedules tasks based on these message tags. However, unlike the static response priorities case, the scheduling is not based on predefined static priorities, it is based on the dynamic due dates of the responses. Those tasks servicing responses that have earlier calculated due dates are favored over those servicing responses with later due dates. The relative functional importance of the response is not explicitly considered although it is reflected in the calculated response time. Using this scheduling algorithm, the results presented in table 6.27 were obtained.

Because of the relative simplicity of the implemented due date calculation scheme, the system and scheduling overheads of table 6.27 do not show any significant increases over those incurred by the baseline system. Also, the number of context switches and preemptions are consistent with the static priority systems (those without time slicing). However, there are significant differences in the response completion data statistics.
<table>
<thead>
<tr>
<th>Scheduling Method</th>
<th>Node Name</th>
<th>Final CPU Time</th>
<th>CPU Usage %</th>
<th>OS Ovhd %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Earliest Due Date</td>
<td>node1</td>
<td>33845</td>
<td>64</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>node2</td>
<td>30077</td>
<td>73</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>node3</td>
<td>37390</td>
<td>82</td>
<td>46</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Sched. Ovhd %</th>
<th>No. of Context Switches</th>
<th>No. of Task Preempts</th>
<th>Avge. RTR Queue Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>1</td>
<td>37</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>node2</td>
<td>3</td>
<td>81</td>
<td>19</td>
<td>2</td>
</tr>
<tr>
<td>node3</td>
<td>3</td>
<td>96</td>
<td>21</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Task Name</th>
<th>Task Priority</th>
<th>Total Ready Time</th>
<th>Msg Queue Size (avg,max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>msg</td>
<td>-</td>
<td>309</td>
<td>(0,2)</td>
</tr>
<tr>
<td>node2</td>
<td>synch</td>
<td>-</td>
<td>6606</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>fault</td>
<td>-</td>
<td>11156</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>audits</td>
<td>-</td>
<td>6143</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>t1mte</td>
<td>-</td>
<td>2930</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>diags</td>
<td>-</td>
<td>3304</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>t1dial</td>
<td>-</td>
<td>532</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>msgdial</td>
<td>-</td>
<td>649</td>
<td>(0,0)</td>
</tr>
<tr>
<td></td>
<td>tonedial</td>
<td>-</td>
<td>1312</td>
<td>(0,0)</td>
</tr>
<tr>
<td></td>
<td>chnlindal</td>
<td>-</td>
<td>867</td>
<td>(0,1)</td>
</tr>
<tr>
<td>node3</td>
<td>tones</td>
<td>-</td>
<td>3735</td>
<td>(0,3)</td>
</tr>
<tr>
<td></td>
<td>cinit</td>
<td>-</td>
<td>4092</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>chnls</td>
<td>-</td>
<td>4173</td>
<td>(0,3)</td>
</tr>
<tr>
<td></td>
<td>stat</td>
<td>-</td>
<td>6766</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>end</td>
<td>-</td>
<td>891</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>connect</td>
<td>-</td>
<td>5372</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>signal</td>
<td>-</td>
<td>3884</td>
<td>(0,1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Response Type</th>
<th>Average Completion Time</th>
<th>Overall Response Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Synch (6) (17)</td>
<td>Fault Report</td>
<td>727</td>
<td>47%</td>
</tr>
<tr>
<td>Synch Loss (14) (43)</td>
<td>Call Setup</td>
<td>4758</td>
<td></td>
</tr>
<tr>
<td>Msg In (23) (57)</td>
<td>Conn. Tone</td>
<td>4814</td>
<td></td>
</tr>
<tr>
<td>Signal Event (3) (100)</td>
<td>Call Takedown</td>
<td>1281</td>
<td></td>
</tr>
<tr>
<td>Audit (5) (20)</td>
<td>Audit Pass</td>
<td>6332</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Changed Synch</td>
<td>1007</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Diag Pass</td>
<td>6998</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.27: Earliest Due Date System Results
In comparing the average response completion times with the predetermined response times, it is apparent that some responses averaged completions within their response times while others did not. The call setup and diagnostic responses averaged completion times of 4758 and 6998 time units respectively which were within their allotted lapse times of 5000 and 12000 respectively. All of the other responses averaged completion times longer than their allotted lapse times. While on the surface this may seem to be quite an unfavorable scenario, a closer look indicates that for most responses, there was no major increase in actual response completion times over the predetermined response times. It is interesting to note that there were no extreme response completion times as have been seen in other implementations (ex. the times for synch loss and synch source change responses in tables 6.19 and table 6.20, the times for diagnostic responses in table 6.21, etc.). All of the response completion times were reasonable in relation to the other scheduling schemes.

The reason for this uniformity is that the earliest due date is a relatively fair scheduling scheme like the first come first served scheme except that EDD is based on "first due date, first served" rather than on "first task ready, first served". There are no extremely large response time deviations for any one type of response as there is generally no "starvation" of one type of response by other responses. Such a fair scheme may not be well suited in applications that have some critical events and some noncritical events. However, this problem can be overcome by assigning due dates based on event importance as was done here by assigning an elongated response time to diagnostic events.

While the EDD scheme does appear to be a very good choice for the telephony application, it does have a few minor drawbacks. From the ready to run time data of table 6.27, it seems that the SYNCH, FAULT, and AUDIT tasks spend large amounts of time on the ready to run queue. This is due to the fact that the due dates for diagnostic, audit and synch source change events have been set to relatively large values compared to the actual execution times required by these responses. For this reason, the tasks involved in these responses are not generally favored for execution. This in itself is not a major concern but the SYNCH and AUDIT tasks are also
responsible for the initial servicing of several interrupts. If they spend large amounts of time on the ready to run queue, larger numbers of latched events will be missed. This is reflected in the event response completion percentage which is down slightly to 47% from its high of 52% in the case of STF scheduling.

6.5.2.4 Least Response Slack Time

In the processing of a response, slack time refers to the amount of time that will remain between the completion of the response and the response’s assigned deadline. The “least (response) slack time” (LST) scheduling algorithm is based on the notion that the next task that should be allowed to run is the one servicing the response that has the smallest slack time. Slack time is truly a dynamic quantity in that its value changes continuously for each active response as the system progresses.

The amount of slack time for each response is calculated using the following formula:

\[
\text{slack} = \text{deadline} - \text{elapsed time} - \text{remaining response processing time}
\]

This concept is illustrated in figure 6.9. In a busy multitasking system, it is unlikely that a complex response will be allowed to complete in an uninterrupted manner. Rather, response execution will be interleaved with other responses. As such, the slack time for every active response needs to be recalculated every time there is an opportunity for a new task to execute because the elapsed time and the time remaining for the completion of the response are continually changing.

The LST scheduling algorithm was implemented in the switching example as presented in sections B.3.7 and B.3.8. In order to implement the LST algorithm prototype, it was necessary to determine the actual times required for the completion of each response. To determine these absolute times, simulations were done in a sequential event response manner (only one response active in the system at a time). From these simulations, the event response times given in table 6.28 were obtained.

As in the case of the EDD system, scheduling data was passed from task to task by means of tags associated with IPC messages. These tags contained the response
Figure 6.9: Slack Time Calculation

<table>
<thead>
<tr>
<th>EVENT</th>
<th>RESPONSE</th>
<th>EXECUTION TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Synch</td>
<td>Change Synch Source</td>
<td>279</td>
</tr>
<tr>
<td>Synch Loss</td>
<td>Report Synch Loss</td>
<td>54</td>
</tr>
<tr>
<td>Call Start Msg</td>
<td>Initiate Call Setup</td>
<td>694</td>
</tr>
<tr>
<td>Connect Tone Msg</td>
<td>Connect Tone</td>
<td>1031</td>
</tr>
<tr>
<td>Diag Request Msg</td>
<td>Diagnostic Pass</td>
<td>1302</td>
</tr>
<tr>
<td>Diag Request Msg</td>
<td>Report Diag Fail</td>
<td>1569</td>
</tr>
<tr>
<td>Integrity Fail</td>
<td>Report Failure</td>
<td>128</td>
</tr>
<tr>
<td>Call Hung Up</td>
<td>Call Takedown</td>
<td>210</td>
</tr>
<tr>
<td>Audit Request</td>
<td>Audit Passed</td>
<td>228</td>
</tr>
<tr>
<td>Audit Request</td>
<td>Report Failure</td>
<td>846</td>
</tr>
</tbody>
</table>

Table 6.28: System Event - Response Execution Times
deadline, the amount of time elapsed in processing the response, and the total amount of time needed to service the response (the values presented in table 6.28). This information was then used by the scheduler to calculate response slack times and schedule the individual tasks accordingly. The results of this prototyping activity are presented in table 6.29.

The LST algorithm has been implemented as a truly dynamic scheduling algorithm in that slack times are calculated at run time by the scheduler. This fact is reflected in the values for operating system overhead and scheduling overhead. This system by far exceeds all others in the amount of CPU time devoted to scheduling functions with node2 and node3 showing 13% and 14% scheduling usage respectively. This overhead is due to the fact that the scheduler must calculate slack times for every task on the ready to run queue each time the currently executing task is blocked. As the number of ready to run tasks increases, so too should the scheduling overhead.

Another interesting aspect of the results of table 6.29 is the large number of task preemptions and the relatively large average ready to run queue sizes. The large number of task preemptions are to be expected in this case as the task priorities (their slack times) are dependent on many factors including the length of time other tasks have control of the processor. For instance, task a may be ready to run with a slack time of 100 while task b is currently executing with a slack time of 80. If task b executes for 30 time units, its slack time will still be 80 but task a's slack time will have been cut to 70. If task b calls a system primitive that causes scheduling action (a nonblocking send, for example), it will be preempted in favor of task a. This type of preemption is absent in the other scheduling schemes as the due dates and task cycle times do not change. These types of overheads can be reduced both in this algorithm and in the EDD algorithm by not using task preemption (as is the usual implementation).

The large ready to run queue sizes are a direct consequence of the large number of task preemptions. When a task is preempted, it does not become blocked. It remains eligible for execution and therefore stays on the ready to run queue. In the LST algorithm, a task can expect to have shorter but more numerous stays in both
<table>
<thead>
<tr>
<th>Scheduling Method</th>
<th>Node Name</th>
<th>Final CPU Time</th>
<th>CPU Usage %</th>
<th>OS Ovhd %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Least Slack Time</td>
<td>node1</td>
<td>44878</td>
<td>53</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>node2</td>
<td>31887</td>
<td>88</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>node3</td>
<td>32498</td>
<td>95</td>
<td>52</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Sched. Ovhd %</th>
<th>No. of Context Switches</th>
<th>No. of Task Preempts</th>
<th>Avg. RTR Queue Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>3</td>
<td>30</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>node2</td>
<td>13</td>
<td>77</td>
<td>66</td>
<td>2</td>
</tr>
<tr>
<td>node3</td>
<td>14</td>
<td>61</td>
<td>52</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Task Name</th>
<th>Task Priority</th>
<th>Total Ready Time</th>
<th>Msg Queue Size (avg,max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>msg</td>
<td>-</td>
<td>642</td>
<td>(0,2)</td>
</tr>
<tr>
<td>node2</td>
<td>synch</td>
<td>-</td>
<td>12381</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>fault</td>
<td>-</td>
<td>6114</td>
<td>(1,3)</td>
</tr>
<tr>
<td></td>
<td>audits</td>
<td>-</td>
<td>8640</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>t1mtc</td>
<td>-</td>
<td>4225</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>diag</td>
<td>-</td>
<td>8115</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>t1diag</td>
<td>-</td>
<td>1682</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>mgdiag</td>
<td>-</td>
<td>1182</td>
<td>(0,0)</td>
</tr>
<tr>
<td></td>
<td>tonedig</td>
<td>-</td>
<td>1973</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>chnldig</td>
<td>-</td>
<td>1029</td>
<td>(0,0)</td>
</tr>
<tr>
<td>node3</td>
<td>tones</td>
<td>-</td>
<td>11758</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>cinit</td>
<td>-</td>
<td>5545</td>
<td>(1,4)</td>
</tr>
<tr>
<td></td>
<td>chnls</td>
<td>-</td>
<td>8101</td>
<td>(0,4)</td>
</tr>
<tr>
<td></td>
<td>stat</td>
<td>-</td>
<td>18147</td>
<td>(0,6)</td>
</tr>
<tr>
<td></td>
<td>end</td>
<td>-</td>
<td>7282</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>connect</td>
<td>-</td>
<td>7420</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>signal</td>
<td>-</td>
<td>12284</td>
<td>(0,1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event Type (# Generated) (%) Completed</th>
<th>Response Type</th>
<th>Average Completion Time</th>
<th>Overall Response Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Synch (5) (20)</td>
<td>Fault Report</td>
<td>874</td>
<td></td>
</tr>
<tr>
<td>Synch Loss (14) (21)</td>
<td>Call Setup</td>
<td>7430</td>
<td></td>
</tr>
<tr>
<td>Mag In (19) (63)</td>
<td>Conn. Tone</td>
<td>5012</td>
<td></td>
</tr>
<tr>
<td>Signal Event (8) (33)</td>
<td>Call Takedown</td>
<td>2279</td>
<td></td>
</tr>
<tr>
<td>Audit (5) (40)</td>
<td>Audit Pass</td>
<td>4548</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Changed Synch</td>
<td>1119</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Diag Pass</td>
<td>9511</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.29: Least Slack Time System Results
the ready to run and running states. This situation is most pronounced in node3 as
the average size of the ready to run queue has gone up to 4 tasks. Usually, longer
stays in the ready to run queue, indicate a lowering of responsiveness.

In examining the response results, it seems that this is indeed the case. In every
case, the average response completion times are worse than those encountered in
the EDD system. The effect is especially pronounced in those responses that go
through node3 including the call setup and tone connect responses. The reason for
this degradation is that the LST algorithm takes a significant amount of processor
time away from the application tasks and it may cause tasks to spend long periods in
the ready to run state (especially on node3). A direct consequence of tasks spending
long periods on the ready to run queue is that they miss a significant number of
latched events. The results show that only 39% of the events were serviced. This
figure could have been even lower except for the fact that the MSG task was able to
handle a large number of incoming message events. This is due to the fact that it
was less busy in handling response completions as the rest of the system was bogged
down.

One final aspect should be noted about the response completion results is that like
the EDD case, there were no wide variations in average response completion times.
Although all of the average responses were completed after their assigned deadlines,
none showed a consistently huge departure from their due dates.

Because the response components of this system take relatively large chunks of
processing time, the LST algorithm does not have a fine enough control over execution
sequences. That is, individual response chunks often take longer than the slack times
of the tasks on the ready to run queue thereby making these waiting tasks late.
As such, the LST scheme is not well suited to this application. However, if more
of the activities required smaller amounts of processing time (had finer granularity),
the LST algorithm may be the best suited. It is very difficult (if not impossible) to
make this type of determination via inspection. Prototyping is by far a more practical
approach. For the telephone switching application, the prototyping activity has shown
that dynamic scheduling does not necessarily have to be extremely computationally

255
expensive and it may be quite beneficial in some cases (the EDD scheme in particular).

6.5.3 Local and Global Scheduling

The final aspect of the scheduling analysis process is the consideration of the use of global scheduling. In systems that use global scheduling, each processor can include a local scheduler which determines whether the deadline of a given task can be met. If it cannot, the task must be scheduled in another processor. This can be done by having a single centralized global scheduler or by having distributed functionality across several processors. The factors that must be considered in making this design decision have been discussed in section 4.1.4. The details on the effects of such a decision are beyond the scope of this thesis but are the subjects of numerous other publications [PBR89, Pil90, Cra88].

A global scheduling scheme is most meaningful when a system is comprised of simple responses (those composed of a single task). Because the telephony example does not fit this criteria, it has not been implemented in this manner. However, it would be a relatively straightforward matter to build such a prototype. In the MDA environment, a single processor could be defined that contained only a single operating system task that performed all of the scheduling operations for all application processors. Communications between the system and application processors would take place through the SHARED construct. As global scheduling generally involves the duplication of tasks across several application processors, the same task declarations could simply be included as part of the required MDA NODE files.

The consideration of global scheduling concludes the process of defining a scheduling scheme that is well suited to the application in question. At the end of this process, most of the major scheduling parameters will have been defined. One of the major advantages of the MDA prototyping process is that unlike other methods, it allows measurements on complete responses in addition to those on individual tasks. This is very important as system behavior is observed in terms of event responses rather than in terms of individual tasks.

The scheduling analysis is the final phase of the design by prototyping method.
At this stage, a final system design should be at hand as it is with the telephone switching example. This systematic design method has brought about a complete design in a relatively short time period. Further, many aspects of the design have already been refined and tested in preparation for final implementation.
Chapter 7

Conclusion

The purpose of this thesis has been to determine the feasibility of developing a software tool and a concrete methodology to assist in the design phase of the real-time system development cycle. The development of a general methodology for this part of the development cycle is difficult because of the wide range of application areas and the numerous possible implementation options.

The functional specification phase of the design cycle has received much attention and as a result, there has been an increasing availability of application oriented specification tools. As such, designers are getting more precise requirements specifications. However, because of the lack of analogous design phase tools and methodologies, they have not been able to substantially improve their final designs. In order to show that the ideas employed in the functional specification phase could be equally applicable in the low level design phase, this thesis introduced a prototyping tool that could be used in a manner similar to an executable functional specification. That is, it could be used to model a proposed system and to observe its behavior through statistical indicators. The identification of problems in the design would then lead to revisions of the model.

Based on this iterative prototyping approach, a generally applicable design methodology was defined by example. This methodology concentrated on four separate areas that must be considered in the design of any real-time multitasking system: software partitioning, choosing a hardware platform, choosing an appropriate IPC scheme, and
the definition of a response scheduler. It was shown that while each of these aspects is an important part of all systems, the choices regarding the details of each can not be generalized and must be made based on application specific factors. Further, it was shown that the effects of the design phase choices on these factors can not fully be determined by inspection nor by a mathematical analysis. Instead, a more realistic approach is to build a prototype of the proposed system and observe its workings.

To this end, a prototyping tool (the Multitasking Design Assist tool) was introduced that was geared specifically towards the modelling of design phase details of multitasking systems. This tool is able to model a functional specification at various levels of design detail and provides the designer with immediate feedback on the consequences of the proposed decisions. Using this tool, a fairly substantial system was able to be modelled along with several system kernels, all in the course of one month. The MDA tool was used to show that proper system partitioning can have significant effects on the communication and operating system overheads incurred by the system. A similar effect was observed in the area of the hardware platform. It was shown that the choice of the number of processors as well as the task to processor allocation had very significant and very specific effects on system responsiveness, communication requirements, and system overheads. With respect to IPC methods, it was demonstrated that several factors must be balanced in making the proper choice of IPC schemes. These included: costs of the individual primitives, the resulting complexity of the application software, and the need for specialized primitives. Finally, the MDA tool was used to do an analysis of scheduling methods that could be used for the example system. It was found that several scheduling methods would have been feasible but the application favored the use of the earliest due date scheme. To the best of the author's knowledge, this was the first time that complete responses and not individual tasks were considered in the scheduling decisions for real-time multitasking systems.

The fact that this tool and its associated design process were able to transform an example requirements specification into a good design specification shows that such tools and concrete methodologies are required in the design phase to complete the
revolution of the development process started by the functional specification phase tools. While the tool and methodology presented in this thesis go to some length in helping to complete the process, there is still room for improvement both in the simulator and in the design phase methodology.

The main purpose of the MDA tool was to provide a structured controlled environment which could be used to model a wide variety of systems at various levels of detail. In order to allow this flexibility, the MDA language included a large number of low level assembly language type instructions along with several higher level "meta" instructions. In addition, it provided a number of base data types along with the capabilities for the definition of user-defined abstract data types. However in building the prototypes, some "workarounds" still needed to be done to overcome the limitations of the MDA language. To overcome the need for such workarounds, a requirement for the following enhancements has been identified:

1. The need for an array structure. In the simulations, the array structure was replaced by using queues of elements. This proved to be quite cumbersome as the queue elements could not be accessed directly but required one by one examination accompanied by the corresponding delq and addq operations. The availability of an array structure would have simplified access of individual elements and would have given truer indications of program overheads.

2. The need for additional boolean operations. The current implementation of the MDA tool allows only the assignment of #true and #false constants to boolean operands. The inclusion of instructions such as and, or, and not (perhaps in a format similar to the arithmetic instructions) would be useful.

3. The ability to have dynamic variable allocation. This facility has been identified as being very useful in the simulations performed as part of this thesis. The main reason for this is that the designer of the kernel software is unaware of a number of system attributes (such as the number of resident tasks) at the time the kernel is coded. If unique structures are required for these tasks, for example, then the designer has no option but to put these structures into some
kind of queue from which they are not directly accessible. If dynamic variable allocation were supported, these structures could be dynamically allocated as directly accessible variables at run-time.

4. The ability to partition activity execution times. The MDA ticks instruction causes a specified number of simulated time units (representing the execution of an activity) to elapse in the simulation. However, this instruction is treated as an indivisible operation. As such, the interrupt system is not checked for the duration of the specified interval. This is a fairly unrealistic situation if some type of preemptive scheduling is in use. The designer can get around this by putting a ticks #1 instruction inside a while loop set for the proper number of iterations. However, this type of workaround should not be necessary.

5. The ability to easily move activities from one task to another. The MDA system provides a very simple means to move tasks between processors but no such facility exists for moving and assigning activities to tasks. Currently, this process involves the complete restructuring of the changed tasks. The system partitioning process could be greatly simplified with this addition.

6. Reduce the simulator's dynamic memory requirements. The current MDA implementation places all of its compiled application code into the main memory of the host machine at all times. While this speeds up the simulation time, it limits the use of the tool to large host machines or to small size simulations. Perhaps a better approach would be to swap dynamically allocated structures between secondary storage and main memory.

7. Allow the saving of simulation states. The MDA system has no means to save the state of a simulation in midstream on a secondary storage device. If this were possible, a simulation could be stopped, saved, and then reloaded and resumed at a later date. This might be useful for very long simulations.

Apart from further update opportunities for the prototyping tool itself, there are a number of issues that need to be addressed before the use of prototyping tools and
methodologies for the design phase gain widespread acceptance. The first of these is to provide a single prototyping tool for all phases of the development process as has been done in multiactivity systems using PAL based tools [KHL88]. A single tool would not only make the designer's task easier in that he or she would only have to learn one system but it would also pay dividends in the accuracies of the prototypes. With a single tool for both the functional and design phases, there would be no chance for any loss of information in the porting of the functional specification from one toolset to another.

While this thesis has focused on four general areas of the system design phase (system partitioning, hardware platform, IPC, and scheduling), it is likely that there are other application specific aspects such as reliability\(^1\) that must be considered for certain classes of systems. These aspects need to be identified and incorporated into the general design methodology. If the design process is not generally applicable, it stands less chance of gaining widespread popularity.

The design methodology presented in this thesis relies on the designer being able to analyze the statistics gathered by the prototyping tool. In many cases, this can become quite an arduous task and mistakes can be made. There is a need to somehow automate this analysis process so that the statistical results can be presented to the designer in a concise preprocessed manner. Depending on the degree of processing, the results should clearly point to the direction that the design should take.

Once all of these issues have been addressed, it is possible that real-time systems design will benefit from the structured design techniques currently enjoyed by other, more established, engineering disciplines.

\(^1\)The MDA tool can be used to build prototypes to model aspects such as system reliability but these were not included in the thesis because of time and especially size considerations.
Bibliography


263


264


Appendix A

Multitasking Design Assist User Guide

This appendix contains the user guide for the Multitasking Design Assist (MDA) tool. As this user guide is meant to be used as a standalone document, the reader may find some of the descriptions of thesis chapter 5 repeated in the following sections. This appendix is a superset of thesis chapter 5.

The MDA tool described in this user guide is used to model uniprocessor or multiprocessor multitasking real-time systems. The MDA tool provides its own modelling language as well as an interpreter on which to run the language. The tool is very flexible in allowing different hardware and software configurations to be modelled as will be described in the following sections.

A.1 MDA Architecture

The MDA tool provides a system composed of one or more virtual machines on which the application and kernel software can run. All software is specified by the user in a fairly high level assembly language (the MDA language) which is interpreted directly by the virtual processors\(^1\).

Each virtual processor contains local memories for the storage of code and data associated with it. In addition, there is a shared memory area which is used for communications between the individual processors. As such, only data items (not code) can be resident in this shared memory. The details of the data storage areas are left to section A.2.2.1. Besides its memory, the other main aspect of the virtual processor is its virtual CPU. It is the virtual CPU that is responsible for simulating the execution of the user programmed application and system software.

A.1.1 Virtual CPU Execution

The execution engine of the MDA simulator is responsible for several things including the actual interpretation of the loaded instructions, determination of which events are to be raised at what times, and the interleaving of a single real processor between a number of virtual processors. The pseudo code for the execution engine is given in figure A.1.

After all of the application and kernel code is loaded into each virtual processor, the simulation can be started. The MDA environment allows any number of processors to be specified in the

\(^1\) In this discussion, the term “virtual processor” is used to describe the processors defined in the system prototype while the term “actual machine” refers to the real machine running the simulation.
Set current virtual processor
WHILE not end of simulation time DO
   Get environment of current virtual processor
   Check for interrupt on current virtual processor
   IF interrupt occurred THEN
      Push PC onto TASKSTK (task stack)
      PC <- Address of interrupt handler
   ENDIF
   Fetch instruction located at PC
   PC <- PC + 1
   Execute instruction
   Increment virtual CPU time by 1
   IF not running locked AND
   not executing statistical instruction THEN
      Increment current virtual processor
   ENDIF
ENDWHILE
END

Figure A.1: Virtual CPU Operation

system. (The maximum number of allowed processors can be configured by the user and would depend on the amount of memory available in the actual machine. See section A.11). If the system consists of more than one processor, then the execution engine simply cycles through all processors executing one instruction per processor per cycle. Each instruction uses one time unit on its virtual processor. There is no overhead associated with changing virtual processors every instruction as the code for all processors is always resident in memory. The advantage of switching processors after every instruction is that a more realistic sequencing of the multiple processor execution stream is achieved.

Because multiple processor execution is interleaved on a single machine in the MDA environment in this manner, it is possible for several processors to access the same shared data item “simultaneously”. To prevent collisions of this type, the MDA language provides a LOCK instruction which operates much like the “test and set” type instruction found on real processors. This instruction essentially locks out other processors from shared memory access until the current access is complete (as signalled by the UNLOCK instruction).

As a result, the only circumstances under which multiple instructions are consecutively executed for a single virtual processor are when there is only one processor in the system, or when the LOCK instruction is encountered. If a processor executes a LOCK instruction, that processor is allowed to continue execution uninterrupted until an UNLOCK instruction is encountered.

The last major function of the execution engine is to check for processor interrupts. MDA allows up to six separate interrupts to be associated with each virtual processor. Each time through the instruction execution cycle, each of the six interrupts associated with the current processor is checked to see if it is active. If an interrupt has been raised, the current program counter is saved
on the TASKSTK and the PC is set to the address of the appropriate user defined interrupt handler code. These interrupt handlers must be coded as part of the processor kernel. The times at which individual interrupts are asserted are determined by the characteristics of the events associated with the interrupts.

A.1.1.1 Virtual Processor Interrupts

Each virtual processor has six interrupt lines associated with it. By default, at the start of the simulation, all of these interrupts are disabled (inactive) and their interrupt handlers are set to be undefined. The event file described in section A.5.4 is used to define these handlers and the characteristics of the interrupts.

As the pseudo code for the execution engine of figure A.1 shows, the interrupts are checked before every instruction execution. However, not all interrupts are checked in each cycle. First of all, only the six interrupts associated with the current virtual processor are eligible to be checked at each cycle. Of these, only one is checked per cycle. Consider a three processor configuration. The six interrupts associated with each of these processors will be checked in the following sequence:

1. Interrupt 0 - Processor 0
2. Interrupt 0 - Processor 1
3. Interrupt 0 - Processor 2
4. Interrupt 1 - Processor 0
5. Interrupt 1 - Processor 1
6. Interrupt 1 - Processor 2
7. Interrupt 2 - Processor 0
8. Interrupt 2 - Processor 1
9. Interrupt 2 - Processor 2
10. Interrupt 3 - Processor 0
11. Interrupt 3 - Processor 1
12. Interrupt 3 - Processor 2
13. Interrupt 4 - Processor 0
14. Interrupt 4 - Processor 1
15. Interrupt 4 - Processor 2
16. Interrupt 5 - Processor 0
17. Interrupt 5 - Processor 1
18. Interrupt 5 - Processor 2
19. Interrupt 0 - Processor 0
20. Interrupt 0 - Processor 1

Thus it takes 18 instruction executions to check and update all of the interrupts in the system and each interrupt must wait 18 time units before it is revisited. Checking and updating the interrupts involves several actions. The first of these is to check whether or not the interrupt is active. If it is inactive, no further updating takes place. If it is active, then its counters are updated based on its interrupt type. After the counters are updated, a decision is made on whether or not the interrupt should be asserted during the current cycle.

For cyclic interrupts, the counter update involves incrementing a counter associated with the event. If the counter has reached its specified period, then the interrupt is asserted and the counter is reset to 0. It should be noted that since this update is done every 18 instruction cycles (for a three processor configuration), the period should be chosen appropriately.

For sporadic interrupts, a random number between 0 and the specified probability is generated each time the event is checked. If the random number is less than the probability, then the event is asserted. Because events are checked relatively often, specified probabilities should be fairly low (1 - 20%) if reasonable delays between interrupt occurrences are expected.

Delayed interrupts are those that are raised after some other specified interrupt has been raised. There are two types of such interrupt, each of which can be specified through the event file. In
VAR int_val, integer;
VAR stk_val, stack;
VAR bool_val, boolean;
VAR q_value, queue;
VAR str, string;

Figure A.2: Example Variable Declarations

the first type, the delayed interrupt is raised a fixed (specified) amount of time after the triggering interrupt. This type of interrupt is implemented in a manner similar to the cyclic interrupt except that the counter is started after the triggering interrupt occurs. In the second type of delayed interrupt, the interrupt occurs a random amount of time after the triggering interrupt. In the MDA interpreter, this random amount of time varies each time the triggering interrupt is raised between 1 and 200 interrupt checks.

The final type of interrupt is the dependent interrupt. This interrupt is raised after a CAUSE instruction is executed that requests triggering of the interrupt in question. The interrupt is raised the next time it is checked after the CAUSE instruction has been executed.

Whenever an interrupt occurs, its occurrence is indicated by incrementing the queue associated with it and recording the virtual CPU time at which the interrupt occurred. If the queue contains the maximum specified number of occurrences then subsequent event occurrences are ignored until an ACKINT instruction has been executed to remove one occurrence from the queue. An interrupt is raised in the virtual machine as long as there is an interrupt occurrence in the associated queue.

### A.2 MDA Variables and Data Types

The MDA system allows data items to be declared as variables and constants just like in most standard high level programming languages. Variables are identified by symbolic names and can be of one of a number of built-in or user defined data types. One thing that must be kept in mind is that while the MDA language allows identifiers in any type of symbolic reference (variables, user defined type names, type field names, file names, etc.) to consist of any type of character (except <space>, '"', ';', or ';') and to be of any length, only the first eight characters are considered significant. For example the identifiers "field.type.1" and "field.type.2" are both valid, they are both stored as "field ty" and refer to the same item. In addition, the MDA language is case insensitive as all input characters are converted to uppercase before compilation.

#### A.2.1 Data Types

There are five base data types in the MDA language: INTEGER, BOOLEAN, STRING, STACK, and QUEUE. Variables can be defined as being any of these types while constants must be one of INTEGER, STRING, or BOOLEAN. Variables must be defined before they are used by means of the VAR directive as illustrated in the examples of figure A.2. The syntax of the VAR directive is as follows:

```plaintext
VAR <variable_name>, <variable_type>;
```

Constants on the other hand, can not be explicitly declared but are coded as instruction operands. A constant value is always preceded by the '"' character. Thus the operand '#-784' represents the constant integer -784.

The INTEGER data type is used to represent whole number values (real numbers are not supported). The size of the integer type is dependent on the size of integers on the machine hosting the simulation. Typically, the valid range of integers will be not less than -32767 to +32768. Only
assignment, arithmetic, arithmetic comparison, and I/O type operations are allowed on integer data types.

The BOOLEAN data type is used to represent the logical values true and false. The internal representation of the boolean type is based on the integer values 0 (false) and 1 (true). As such, a boolean type requires the same amount of space as an integer. However, the MDA language does not allow the direct integer assignment of the values 0 or 1 to boolean variables. The only values that can be assigned to boolean variables are the constants #true and #false. The only operations allowed on boolean types are assignment, logical comparison, and I/O.

The STRING data type is the same as in any other language except that the characters in a string constant are delimited by the '"' character. The operand #/string constant/ is an example of a valid string. Strings are limited to a maximum of 39 characters. The operation types valid for strings are assignment, comparison, and I/O. In the case of comparison, strings can only be reliably compared for equality or inequality. If arithmetic comparisons are attempted on string data types, the results will be unpredictable. As will be discussed shortly, string types are instrumental in indirect addressing.

The STACK and QUEUE types do not have any constant representations and exist as variables only. The stack is simply a LIFO list, each element of which is capable of storing elements of any data type. That is, a single stack may contain elements of different types. The elements of a stack are dynamically allocated so there is no preset upper bound on the size of the stack. The only limitation is the amount of memory available on the actual machine. The queue data type is simply the FIFO equivalent of the stack. Only stack type operations such as push, pop, and sizes are allowed on the stack data type and only queue type operations such as addq, delq, and sizeq are allowed on the queue data type. The queue data type is used in place of conventional arrays.

A.2.1.1 User Defined Types

In addition to the five base built-in data types, the MDA language allows the user to define his or her own abstract data types through the use of the TYPE directive. These complex data types use the five base types to build record aggregates to better organize related data. An example of such an aggregate is the task control block maintained by the kernel. The type definition of such a control block along with its corresponding variable declaration is given in figure A.3.

The general form of the TYPE directive is:

```
TYPE <type_name> = <field1>:<field1_type>, ..., <fieldn>:<fieldn_type>;
```

If there are too many fields to comfortably fit on a single input line, the type declaration can span multiple lines using the "\*" as a continuation character. The "\*" can be used in place of any of the commas separating the individual fields of the type. This is illustrated as follows:

```
TYPE <type_name> = <field1>:<field1_type>, <field2>:<field2_type> *
      , <field3>:<field3_type> *
      , ...
      , <fieldn>:<fieldn_type>;
```

Note that each input line must end with a semicolon even in the case where the type definition spans more than one line.

The individual fields within the aggregate record are accessed as in most high level languages by separating the qualifiers with periods as in: curr_tcb.run.data.rdy.time. This allows access to the "rdy_time" field of the task control block. A user defined data type can be nested to any level and can use any of the five base types or other user defined types in its declaration. The only operation that can be performed on a complete aggregate is the assignment operation (with the move instruction). For one complex variable to be assigned to another, they must both be of the same user defined type. The operations that can be performed on the individual fields making up the user defined type are limited only by the type of the field itself.

While the user defined type can be used to define virtually any type of abstract data type, the user must be careful when including a queue or stack type as a field of the aggregate record. In most cases there is no problem in this situation except when such an aggregate is assigned as a complete
Figure A.3: Example User Defined Type

block or when such an aggregate is added to a queue or stack. The MDA language does not allow queue or stack variables to be directly added as elements of other stacks or queues. That is, a queue type element can not be added to another queue for example. However, user defined types can be added as elements to queues. If the user defined type contains a stack or queue field, it is still syntactically valid. However, it is up to the user to explicitly manage this embedded queue. For example when an assignment operation is done on a structure containing a queue, only the pointer to the front of the queue is copied so that after the assignment, both structures will point to the same queue. This is not the case for moves on first level queues where a new queue is created and all of the elements are automatically copied. In cases such as these, it is advisable to perform the complete aggregate assignment and then to separately perform the assignment of the qualified queue field.

A.2.2 Variable Definition

As mentioned in the previous section, the VAR directive is used to declare data items. This directive performs two functions: it makes the name and type of the data item known to the MDA system, and it reserves the required amount of space for the data item. The place where this space is reserved depends on where the VAR directive is encountered. The VAR directive can be found in one of three places: within a node file, within a kernel file, and within a task file. These files are described in detail in later sections. At this point, it is sufficient to state that the node file describes all data items global to a virtual processor and a task file describes all data local to a particular task within a virtual processor. The data declared within a kernel file is for use by the kernel software on a particular virtual processor.
Like the VAR directive, the STAT directive can be used to declare data items. The STAT directive can be used anywhere that the VAR directive is used and it functions in exactly the same manner. The only difference between the two is that the STAT directive is used to declare data items whose values need to be examined at the end of the simulation. These variables can act as statistics gatherers programmed by the users. At the end of a simulation, the MDA interpreter records the values of all STAT variables and copies them to an output file. Any user defined or built-in type variable may use the STAT directive.

The final directive used to declare variables is the SHARED directive. Again, the SHARED directive behaves in exactly the same manner as the VAR directive with one exception. This exception deals with the location at which space for the declared variable is reserved. Variables declared with the SHARED directive are allocated in the shared memory used to interlink the virtual processors of the system. With these various levels of memory allocation, there is a need for some type of scope rules to arbitrate their access.

### A.2.2.1 Scope Rules

Along with its memory requirements for code space, each task also has private data associated with it. This data is accessible only to the task in which it is defined. As each task is loaded into the virtual machine, space is allocated for each of the variables defined within the task. Each task is allotted its own local data space. The MDA system does not provide any facilities for dynamic memory management. As such, all data must be defined in a static fashion prior to use. During execution, a context switch automatically changes the task data set that is accessible to the newly executing task. Context switches are performed by the CXTSW instruction of the MDA language.

Like the application tasks, kernel code is user-programmed in the MDA language. The kernel consists of the instructions used to define the operating system of the virtual processor. However, unlike the application code, there is only one kernel per processor. All data associated only with the kernel is treated as local data for the kernel task. Although the kernel variables are allocated in the same place as the virtual processor's global variables, by convention this data should only be accessed by the code included as part of the kernel.

The final memory area is allocated to data that is global to the processor. All tasks as well as the kernel can access this data at any time without the need for any type of context switching. Data is considered to be global if it is defined outside the syntactic scope of any task. Any type of data, be it global or local is defined in terms of named variables and constants. If the same name is used for a data item local to a task and for a global data item, the local item is accessed when the common name is encountered within that task.

The interlink scheme between the individual virtual processors is quite straightforward in that the underlying structure is simply a single shared memory. Data items can be placed in this shared memory by one processor and accessed by the other processors. The only restriction is that all data items to be placed in the shared memory must be defined (with the SHARED directive) before they are used. This allows the MDA tool to allocate a fixed area of shared memory for each shared variable prior to the start of execution.

The access rules for shared memory are similar to those for global data except that any task on any processor in the system can access the shared memory at any time. For any variable access, the task's local data is searched first, followed by global processor data, followed finally by the shared data. As a result, a shared structure can be accessed only if no structure of the same name exists in the accessing task's local or global data space.

In summary, there are three scopes defined in the MDA environment as illustrated in figure A.4: local, global, and shared. No two identifiers may have the same name in the same scope regardless of whether they are variables or fields or types. However, if they are declared in different scopes, they may have the same name and the above described scope rules will be used to access them.

### A.2.2.2 Predefined Variables

Along with the variables defined by the user, there are several variables that are predefined and preinitialized by the MDA system. They are used to facilitate the transfer of information from the run-time environment to the user-written application and kernel software. These six special global
*system variables* are predefined in each of the virtual processors and initialized by the MDA loader. The functions of these variables is outlined in the following list.

1. **TASKSTK** - This stack variable represents the stack associated with each task in the processor. There is one stack for each task. When a context switch occurs, the stack of the old task is saved and the stack of the new task is assigned to the TASKSTK variable.

2. **CURRTASK** - This is a string variable that contains the name of the task that is currently executing.

3. **TASKSTRT** - This variable contains the start address of the currently executing task. Because the MDA language requires that all subroutines be defined before use, the start address of the mainline of a task is not known until the task has been loaded into memory.

4. **CURRPRIO** - This integer variable provides the current priority of the presently executing task.

5. **TASKLIST** - This queue variable contains the name of every task resident in the processor. The names are queued in the order that the tasks were loaded.

6. **PC** - This integer variable is the program counter associated with the currently executing task. It contains the address of the next instruction to be executed in the task. When a context switch occurs, the PC of the old task is saved and the program counter of the new task is assigned to the PC variable.

The TASKSTK, CURRTASK, TASKSTRT, CURRPRIO, and PC variables are automatically updated to contain the information associated with the newly executing task after every context switch. All of these six variables are declared in the global scope of each virtual processor. As such, they can not be redefined in that scope. However, they may be redefined within the local scope of an application task. If this is done, it will be impossible for the application task to access the predefined variables as the scope rules will access the local variables instead.
IMMEDIATE : #744
# / This is a string /

DIRECT : curr_tcb.run_data.run_time
         rtr
         tcb_name

INDIRECT : @tcb_name
           @tcb_name.priority

Figure A.5: MDA Addressing Modes

A.3 Data Addressing Modes

The data items declared through the VAR, STAT, and SHARED directives can be accessed using any one of three addressing modes: immediate, direct, or indirect. When accessed in any one of these ways, they can act as instruction operands. There are no restrictions as to which addressing mode can be used with which MDA instruction.

The immediate mode simply refers to constant operands which can be of type integer, boolean, or string. The direct mode refers to variables of any type defined through the VAR, STAT, or SHARED directives. Direct operands have no special character prefix and they may be qualified if they are of a user defined type.

Indirect operands are variables that serve as pointers to other variables. However, rather than containing the storage address of the dereferenced variable, indirect operands contain the symbolic name of the variable to which they are pointing. As a result, all indirect operands must be of type string. Indirect operands are prefixed by the '@' character. For example, assume that curr_task.stats.rdy.time is a valid direct variable reference to the curr_task variable which is of a user defined type. This data item could also be referenced by the expression @tcb_name.stats.rdy.time provided that tsk_name was a string variable which contained the value "curr_task". The indirect addressing mode is made more powerful by the fact that the pointer itself can contain any level of qualification for referencing user defined structures. That is, in the previous example, the same field could have been referenced by the expression @tsk_name.rdy.time if the tsk_name string contained the value "curr.task.stats". An example of each type of addressing mode is given in figure A.5.

A.4 The MDA Language Instruction Set

The MDA language is similar to many mnemonic based assembler languages with three significant exceptions. The first is the inclusion of high level control constructs similar to those found in languages such as C or Pascal. The second is the inclusion of powerful "meta" instructions that allow high level operations on abstract data types or complex system operations such as context switches. The final deviation from standard assembler languages is that the MDA language directly supports the definition and access of abstract data types.

All of the MDA instructions have the format:

    opcode  operand1, ..., operand4; <comment>

The number of operands varies between zero and four. The purpose of these instructions is to program the details of activities, procedures, system primitives, and task mainlines. The instructions can be divided into seven distinct classes. These classes and their constituent instructions are described in the following subsections.
A.4.1 Assignment Class

The assignment class of instructions is used to move data items from one variable to another or to move constants into variables. The only instruction included in this class is the move instruction.

**MOVE Instruction**

*FORMAT:* move opnd1, opnd2; <comment>

This instruction copies the contents of opnd1 to opnd2. The two operands may be of any type as long as they are of the same type. One further restriction is that opnd2 can not be a constant. For user defined types, entire structures or parts of structures may be assigned using one move operation as long as the structures or substructures are of the same user defined type.

For arrays and stacks, opnd2 is first purged (all elements removed) and then every element of opnd1 is pushed (in the case of stacks) or queued (in the case of queues) onto opnd2 so that at the end of the operation, the two queues contain the same elements in the same order.

A.4.2 Arithmetic Class

The arithmetic class of instructions is used to perform numerical operations.

**ADD Instruction**

*FORMAT:* add opnd1, opnd2, opnd3; <comment>

The add instruction performs the following operation:

opnd3 <- opnd1 + opnd2

All operands must be of type integer and opnd3 can not be a constant.

**SUB Instruction**

*FORMAT:* sub opnd1, opnd2, opnd3; <comment>

The sub instruction performs the following operation:

opnd3 <- opnd1 - opnd2

All operands must be of type integer and opnd3 can not be a constant.

**DIV Instruction**

*FORMAT:* div opnd1, opnd2, opnd3; <comment>

The div instruction performs the following operation:

opnd3 <- opnd1 / opnd2

All operands must be of type integer and opnd3 can not be a constant. The result assigned to opnd3 is an integer. Any remainder is ignored.

**MULT Instruction**

*FORMAT:* mult opnd1, opnd2, opnd3; <comment>

The mult instruction performs the following operation:

opnd3 <- opnd1 * opnd2

All operands must be of type integer and opnd3 can not be a constant.

**RAND Instruction**

*FORMAT:* rand opnd1, opnd2; <comment>

The rand instruction generates a random integer value between 0 and opnd1 - 1 and assigns it to opnd2. Both operands must of type integer and opnd2 can not be a constant. If opnd1 <= 0, then a value of 0 is assigned to opnd2.
A.4.3 Stack Class

The stack class of instructions is used to maintain LIFO stacks.

**PUSH Instruction**

**FORMAT:** push \(<opnd1>, <opnd2>; <comment>\)

This instruction pushes the value of opnd1 onto the top of the stack specified by opnd2. Opnd1 can be any type while opnd2 must be of type stack. There is no limit to the number of items that can be pushed onto a stack (except the host machine’s memory size). A stack can contain elements of any type and even of different types.

**POP Instruction**

**FORMAT:** pop \(<opnd1>, <opnd2>; <comment>\)

The pop instruction removes the top item from the stack specified by opnd1 and assigns it to opnd2. Opnd1 must be of type stack while opnd2 must be of the same type as the element being popped from the stack. Because a stack can contain any type of element (possibly different types in the same stack), it is up to the user to ensure that the element being popped will be of the same type as the receiving variable (opnd2). Also, opnd2 can not be a constant. In addition, the user must ensure that the stack is not empty before performing the pop operation. Otherwise, a run time error occurs.

**SIZES Instruction**

**FORMAT:** sizes \(<opnd1>, <opnd2>; <comment>\)

The sizes instruction is used to determine the number of elements contained in the stack specified by opnd1. This integer value is assigned to opnd2. Opnd1 must be of type stack and opnd2 must be of type integer. Opnd2 can not be a constant.

A.4.4 Queue Class

The queue class of instructions is used to maintain prioritized queues.

**ADDQ Instruction**

**FORMAT:** addq \(<opnd1>, <opnd2>, <opnd3>; <comment>\)

This instruction adds the value of opnd1 to the end of the queue specified by opnd2 at the priority specified by opnd3. Opnd1 is not altered in any fashion. Opnd1 can be of any type while opnd2 must be of type queue and opnd3 must be of type integer. There is no limit to the number of items that can be added to a queue (except the host machine’s memory size). A queue can contain elements of any type and even of different types.

When an item is added to a queue, opnd3 is examined to determine the position of the new element. Elements with higher priority (higher values of opnd3) are placed closer to the front of the queue. Items to be added with the same priority as existing queue elements are added behind the already existing queue elements. Thus a standard FIFO queue can be attained by always adding elements using the same value for opnd3.

**DELQ Instruction**

**FORMAT:** delq \(<opnd1>, <opnd2>; <comment>\)

The delq instruction removes the first item from the front of the queue specified by opnd1 and assigns it to opnd2. Opnd1 must be of type queue while opnd2 must be of the same type as the element being removed from the queue. Because a queue can contain any type of element (possibly different types in the same queue), it is up to the user to ensure that the element being removed will be of the same type as the receiving variable (opnd2). Also, opnd2 can not be a constant. In addition, the user must ensure that the queue is not empty before performing the delq operation. Otherwise, a run time error occurs.

**SIZEQ Instruction**
FORMAT:  sizeq <opnd1>, <opnd2>; <comment>

The sizeq instruction is used to determine the number of elements contained in the stack specified by opnd1. This integer value is assigned to opnd2. Opnd1 must be of type queue and opnd2 must be of type integer. Opnd2 cannot be a constant.

A.4.5 System Class

The system class of instructions is quite varied and contains instructions used for task management, interrupt management, and control of simulation time.

CTXSW Instruction

FORMAT:  ctxsw <opnd1>; <comment>

The context switch instruction serves to change the application task that gets access to the processor. Opnd1 specifies the name of the task that is to take control of the virtual processor. As such, opnd1 can be a string constant, string variable, or simply an identifier corresponding to the name of a defined task. For example, if a task has the name task1, then the instructions:

```c
ctxsw task1;
ctxsw #/task1/;
```

are equivalent. The only stipulation for a string variable as opnd1 is that it must contain the name of a valid task when the ctxsw instruction is executed. Otherwise, a run time error occurs.

The execution of the ctxsw instruction performs the following actions:

1. The variable CURRTASK is set to the name of the newly active task,
2. The variable CURRPRIO is set to the priority of the newly active task,
3. The variable TASKSTRT is set to the start address of the newly active task,
4. The variable TASKSTK is set to the automatically saved stack of the newly active task,

LOCK Instruction

FORMAT:  lock; <comment>

The lock instruction does not have any operands. Its purpose is to prevent the execution engine of the interpreter from interleaving virtual processor execution. When a virtual processor executes this instruction, its instructions are executed contiguously without interference from other virtual processors.

UNLOCK Instruction

FORMAT:  unlock; <comment>

This instruction serves to undo the action of the lock instruction. When the unlock instruction is executed, the execution engine resumes interleaving instruction execution between all of the defined virtual processors.

CAUSE Instruction

FORMAT:  cause <opnd1>, <opnd2>; <comment>

The cause instruction causes the assertion of a specified interrupt on a specified processor. The name of the processor on which the interrupt is to be raised is specified in opnd1. As such, opnd1 must be of type string and must contain the name of a defined node. The interrupt number that is to be raised in specified by opnd2. As such, it must be of type integer. This interrupt number must be in the range 0-5 corresponding to the interrupt range and it must refer to an interrupt that has been defined as being of the dependent type. Otherwise, a run time error is generated.

ACKINT Instruction
FORMAT:  ackint <opnd1>; <comment>

This instruction serves to acknowledge an interrupt. The interrupt number to be acknowledged is specified in opnd1. Opnd1 must be of type integer and its value must be in the range 0-5, corresponding to the valid interrupt number range. For interrupts which have current queue sizes greater than 0, this instruction simply decrements the number of pending interrupts by 1. If there are no pending interrupts, then no action is taken.

HALT Instruction

FORMAT:  halt; <comment>

This instruction simply causes execution to stop on all virtual processors. It is treated as an error condition and results in the termination of the simulation.

ENABLE Instruction

FORMAT:  enable <opnd1>; <comment>

This instruction is used to activate a specified interrupt. The interrupt number to be activated is specified by opnd1. As such, opnd1 must be of type integer. If the value of opnd1 is greater than the highest interrupt number (5), then all interrupts on the processor executing the instruction are activated. The action of the enable instruction is delayed by one instruction execution cycle on the processor in question. For example, in the sequence:

   enable #10;
   move #true, bool_var;

interruption may be generated only after the move instruction has completed execution. This is to allow system primitives to return execution to the calling task safely.

DISABLE Instruction

FORMAT:  disable <opnd1>; <comment>

This instruction is used to deactivate a specified interrupt. The interrupt number to be deactivated is specified by opnd1. As such, opnd1 must be of type integer. If the value of opnd1 is greater than the highest interrupt number (5), then all interrupts on the processor executing the instruction are deactivated.

BIND Instruction

FORMAT:  bind <opnd1>, <opnd2>; <comment>

The bind instruction is used to associate a kernel primitive interrupt handler with a specified interrupt number. Opnd1 is used to specify the name of the primitive and must be of type string. Opnd2 is used to specify the number of the interrupt to be associated with the primitive. As such, it must be an integer type in the range 0-5. After this instruction is executed, whenever the specified interrupt is raised, the specified primitive will be called.

TICKS Instruction

FORMAT:  ticks <opnd1>; <comment>

The ticks instruction is used to simulate a fixed time delay. It simply increments the time clock of the virtual processor that executes it by the specified amount of time units. The number of time units is specified in opnd1. As such, it must be of type integer.

BRKPT Instruction

FORMAT:  brkpt; <comment>
This instruction interrupts the execution of the virtual processors and transfers control to the MDA runtime debug environment which allows single stepping, variable examination, etc. (See section A.9 for details about this environment).

**TIME Instruction**

**FORMAT:** time <opnd1>; <comment>

The time instruction notes the virtual CPU time of the processor on which it is executed and copies it to the variable specified by opnd1. As such opnd1 must be of type integer and it can not be a constant. The time instruction is used to obtain time stamps.

**ETIME Instruction**

**FORMAT:** etime <opnd1>, <opnd2>; <comment>

This instruction operates much like the time instruction except that instead of returning the current virtual CPU time it returns the virtual time at which the next pending interrupt event was originally raised. This instruction must be called before the corresponding acknt instruction to obtain the interrupt occurrence time. Opnd1 specifies the number of the interrupt and opnd2 specifies the variable that will be used to store the time. As such, opnd1 must be an integer in the range 0-5 and opnd2 must be an integer variable (not a constant). If there are no interrupts pending, the time at which the last interrupt was raised is returned (0 if it was never raised).

### A.4.6 Control Class

The control class of instructions is used to alter the flow of execution within an MDA program either by procedure calls or logical constructs.

**CALL Instruction**

**FORMAT:** call <opnd1>, <opnd2>, ..., <opndn>; <comment>

This instruction executes a procedure call. The name of the procedure to be called is specified by opnd1. As such this operand must be the name of a procedure that has been defined prior to the compilation of the call instruction. It should be noted that the call instruction may also be used to call primitives but ONLY IN THE KERNEL FILE. If application tasks need to call kernel primitives, they must use the CALLP directive as indicated in section A.5.3. Note that opnd1 is NOT a string constant or variable but the actual name of the procedure. The other operands specify the parameters to be passed as part of the procedure call. These parameters can be of any type. Parameters can be passed by value or by reference. To pass by reference, the name of the variable to be referenced is passed as a string parameter and then used in indirect addressing by the called procedure.

The actions taken by the call instruction are as follows:

1. Push the current program counter onto the stack (TASKSTK),
2. Push the specified parameters onto the stack (TASKSTK) in order of appearance. That is, opnd2 is pushed first and opndn is pushed last,
3. Set the program counter (PC) to the address of the first instruction of the specified procedure.

It should be kept in mind that the called procedure should pop the parameters from TASKSTK in the reverse order that they were pushed by the call instruction. Also, the called procedure must not pop the return address off the stack (unless this is the actual intention).

**RET Instruction**

**FORMAT:** ret; <comment>
The return instruction can be used anywhere in a called procedure or primitive to return control to the calling code. This instruction pops the return address from the TASKSTK and sets the program counter to this location. The called procedure must be careful to pop all parameters off the TASKSTK before executing the ret instruction. If an ENDPROC or ENDPRIIM directive is encountered before a ret instruction, it has the same effect as a ret instruction.

IF..ELSE..ENDIF Instructions

**FORMAT:**

```plaintext
if <logical expression>; <comment>
    <instructions>; <comment>
else;
    <instructions>; <comment>
endif;
```

**OR**

```plaintext
if <logical expression>; <comment>
    <instructions>; <comment>
endif;
```

The if construct is used in exactly the same manner as in high level languages. It may include an optional else clause. If the logical expression evaluates to "true", the instructions within the body of the if clause are executed. Otherwise the else clause is executed if it is present.

A logical expression can consist of operands separated by the following arithmetic comparison operators: >, >=, <, <=, =, <> or by the following logical operators: &., |, ~ which represent the logical operations and, or, and not respectively. Arithmetic comparisons can be done only on integer or string data types and the logical operations can only be done on boolean data types or on arithmetic comparison subexpressions that evaluate to true or false. The following are two examples of valid logical expressions:

```plaintext
((tcb.prio >= currprio) & (size = #27))
```

```plaintext
(running) & (~error)
```

The results of any logical expression must be able to take on the values #true or #false. In the second example above, the variables "running" and "error" must be declared to be of type boolean.

The if construct can be nested to any level and can contain any number and nesting levels of while constructs as well.

**WHILE..ENDWHILE Instructions**

```plaintext
FORMAT:
while <logical expression>; <comment>
    <instructions>; <comment>
endwhile;
```

The while construct defines the looping construct in the MDA language. The instructions between the while and endwhile instructions are repeatedly executed in sequence while the logical expression evaluates to true. (See the discussion on the if instruction for a description of logical expressions). The logical expression is always evaluated at the top of the loop. The while construct can be nested to any level and may contain any number and nesting levels of if constructs as well.

### A.4.7 I/O Class

The I/O class of instructions is used to interface between an executing MDA simulation and the outside world through terminal input and output and through file output.

**READ Instruction**

**FORMAT:**

```plaintext
read <opnd>; <comment>
```

282
The read instruction is used to do input from the terminal. Execution of this instruction causes the simulation to wait for input from the terminal once carriage return \(<CR>\) is entered, the value entered is assigned to the variable specified by opnd1 and execution continues. Opnd1 must be a variable (not constant) of type string, integer, or boolean. The value entered at the terminal must correspond to the type of opnd1. If opnd1 is of type string, then any character sequence of up to 39 characters may be entered. In opnd1 is an integer then a numeric value preceded by an optional + or - sign may be entered. For boolean types, the input must be either true or false.

**PRINT Instruction**

**FORMAT:**\[\begin{align*}
\text{print} & \quad \text{<opnd1>}; \quad \text{<comment>}
\end{align*}\]

The print instruction causes the value of opnd1 to be printed to the terminal screen. Opnd1 can be a variable or constant of type string, integer, or boolean. For boolean items, the values true or false are printed.

**PRINTCR Instruction**

**FORMAT:**\[\begin{align*}
\text{printcr} & \quad \text{<opnd1>}; \quad \text{<comment>}
\end{align*}\]

The printer instruction is exactly the same as the print instruction except that a carriage return is printed at the end of the output item.

**FPRINT Instruction**

**FORMAT:**\[\begin{align*}
\text{fprint} & \quad \text{<opnd1>}; \quad \text{<comment>}
\end{align*}\]

The fprint instruction is analogous to the print instruction with two exceptions. The first is that instead of printing the value of the specified operand to the terminal screen, it is copied to the output statistics file specified at the start of the simulation. The second exception is that opnd1 is not limited to string, integer, and boolean types. Opnd1 can be of any type including user defined types, stacks, and queues. For user defined types, the entire contents of the data structure are dumped to the output file in hexadecimal format. It is up to the user to identify the individual fields of the structure. In addition to the value contained in the structure, its size and user defined type are printed (the sizes and types are printed for all types of operands). For stacks and queues, each element of the stack or queue is individually printed regardless of whether it is an integer, string, boolean, or user defined item. Also, its position within the stack or queue is printed. If the stack or queue is empty, a message to that effect is printed to the output file.

**FTYPE Instruction**

**FORMAT:**\[\begin{align*}
\text{ftype} & \quad \text{<opnd1>}; \quad \text{<comment>}
\end{align*}\]

The ftype instruction is completely analogous to the print instruction except that it prints to the user specified output statistics file instead of to the terminal screen. The ftype instruction is used instead of the fprint instruction when output formatting is required and when extra information such as data size and type are not required. Like the print instruction, opnd1 is limited to integer, string, and boolean types.

**FTYPECR Instruction**

**FORMAT:**\[\begin{align*}
\text{ftypecr} & \quad \text{<opnd1>}; \quad \text{<comment>}
\end{align*}\]

The ftypecr instruction is exactly the same as the ftype instruction except that a carriage return is printed at the end of the output item.
A.5 MDA Input Files

There are five types of input files required by the MDA system as well as two types of output files produced by MDA. The input files are used by the user to specify the description of the system to be modelled by MDA. It is these files that contain all of the MDA language instructions to be executed in the virtual machines, the specification as to the number of virtual processors to include in the model, the application tasks and kernels that each processor should contain, and the number and types of events to be raised in each of the processors. The output files provide listings of compiled input files and also provide statistics on the simulation run. The input files are the system descriptor file, the node file, the kernel file, the task file, and the event file. The output files are the listing files and the statistics file.

The only real common feature to all of the input files is that each statement in the file must end in a semicolon (;) and that any comments can follow the semicolon. Also, no statement except the TYPE directive may span more than one input line and blank lines are allowed in any input file. Finally, if the MDA tool is used in the UNIX environment, all input file names must be in lowercase letters. In the DOS environment, the case of the input file names is not important. The individual files are detailed in the following sections.

A.5.1 System Descriptor File

The system descriptor file can have any name but by convention, it ends in the suffix .sys. The .sys file contains only the NODE directive. This file is used to specify the number and symbolic names of all virtual processors that are to be included in the system. The names of the processors also must be the names of the node files associated with each. For example, a file named example.sys may contain the following lines:

    NODE cpu1;
    NODE cpu2;
    NODE dsp;

This signifies that there are to be three processors with names cpu1, cpu2, and dsp corresponding to virtual processors 0, 1, and 2 respectively. It also indicates that the specifications for these three nodes are contained in files cpu1.nod, cpu2.nod, and dsp.nod respectively.

A.5.1.1 Node File

All node file names must end with the suffix .nod. The .nod files are used to specify five things associated with each virtual processor. These are: the kernel to be used in the processor, user defined types global to the processor, the global variables, the events (interrupts) associated with the processor, and the tasks resident on the processor. The syntax of the node file is given in figure A.6.

The TYPE directives are used to define the global user defined types that are to be visible to all tasks in the processor. Similarly, the VAR directives define the global data items that are to be accessible to all tasks in the processor. Both the TYPE and VAR directives are optional in the node file and the VAR directive may be replaced or freely intermixed with the STAT directive as described in section A.2.2.

The EVENT directive specifies the name of the file that contains the event (interrupt) specifications for this processor. For example, if the node file included the lines:

    EVENT clock;
    KERNEL base;
    TASK task1, 50;
    TASK task2, 10;

then the event file would be named clock.evt. The EVENT directive is optional and may be omitted if there are to be no interrupts associated with the virtual processor;
| NODE FILE SYNTAX |

TYPE <type definition>;

TYPE <type definition>;

VAR <variable name>, <variable type>;

VAR <variable name>, <variable type>;

EVENT <event file name (no suffix)>;

KERNEL <kernel file name (no suffix)>;

TASK <task name>, <task priority>;

TASK <task name>, <task priority>;

ENDNODE;

Figure A.6: Node File Syntax

The KERNEL directive is used to provide the name of the kernel to be loaded into the virtual processor. In the example, it indicates that the specifications for this kernel are contained in a file named base.knl. There must be one KERNEL directive in each node file.

The TASK directives are used to specify the names and priorities of every application task that is to be resident in the processor. The task names specified in these directives must correspond to the names of the files that contain the individual task specifications. In the example, these files would be named task1.task, and task2.task. The tasks would have priorities of 50 and 10 respectively. There must be at least one TASK directive in each node file. Finally, each node file must end with the ENDNODE directive.

A.5.2 Kernel File

All kernel file names must end with the suffix .knl. A processor kernel defined in the MDA system consists of a number of elements. The entire kernel is syntactically contained within a single task and consists of local kernel data, definition of shared data items, definition of subroutines, definition of system primitives, interrupt handlers, and the task mainline code. These subroutines, system primitives, interrupt handlers, and task mainline are coded in the instructions of the MDA language. The syntax of the kernel file is as follows:

KERNEL <kernel name>;

TASK <kernel task name>, <priority>;

TYPE <kernel user defined type declaration>;

TYPE <kernel user defined type declaration>;

TYPE <kernel user defined type declaration>;

VAR <kernel variable declaration>;

285
The first two statements of this file must be the KERNEL and TASK directives. They serve the same functions as in the .nod file except that they do not reference any subsequent files. The KERNEL directive serves to identify the name of the kernel and the TASK directive merely gives the name and priority of the kernel task. These directives may be followed by any number of TYPE and VAR directives that specify the user defined types and data items local to the kernel. In addition, the kernel may include one or more SHARED directives. These directives are similar to VAR except that they define data items that are to be resident in shared memory and thus are accessible by any task on any processor. All of the TYPE, VAR, and SHARED directives are optional. The only stipulation is that the TYPE directives must come before any VAR or SHARED directives. The positions of the VAR and SHARED directives can be intermixed among themselves and as always, the VAR directive can be replaced by the STAT directive.
The local kernel data defined by the VAR directives consists of items such as the control blocks of all the application tasks, possibly IPC buffers, possibly semaphore structures, etc. These items are accessible only to the kernel and are used for system related functions. All of this data is defined by the user. The other class of data definition within the kernel is the definition of shared data. This is the data that is accessible to any task on any processor in the system regardless of the kernel in which it is defined. Shared data variables can be defined in the kernel of any processor. If two kernels define a shared data item with the same name, a warning is issued and only the definition contained in the kernel loaded first remains valid.

The type and variable declarations are followed by the definitions and code of local procedures and system primitives. There is no restriction as to the numbers of procedures and primitives that can be coded and in fact, it is not necessary to have any procedures or primitives. The only stipulation is that all procedure definitions must precede all primitive definitions.

The main bodies of procedures and primitives contain the MDA language instructions. All procedures must start with the PROC directive that specifies the name of the procedure and they must end with the ENDPREC directive. ENDPREC serves the same function as the ret instruction. All procedures must be defined before they are used. Any procedures defined within the kernel task can be called only by the kernel task mainline, by other procedures defined within the kernel task, and by primitives defined within the kernel task.

Primitives are very similar to procedures with three notable differences. The first difference is that the PROC and ENDPREC directives of the procedure are replaced by the PRIM and ENDPREC directives even though they serve exactly the same purposes. The second difference is that primitives are well known throughout the processor and can be called from any application task resident on the same processor. The third difference is that primitives can be defined only in the kernel task. Kernel primitives can be defined for two reasons. The first is that they are to act as functions that can be called by application software to perform operations such as intertask communication. The second reason is that they are to act as interrupt handlers. Only primitives can be used as interrupt handlers. Once an interrupt is raised, control is transferred to its associated primitive. Thus all interrupts must first be channelled through the kernel. In this way, the kernel has control over things such as which task(s) should be readied as a result of the interrupt, what data should be saved, and whether task preemption is warranted.

Finally, the mainline of the kernel task is used primarily to initialize system data as it is the first user coded software that runs on processor startup. The mainline is delimited by the BEGIN and ENDTASK directives. The mainline can contain any MDA instructions and can be of any length. One of the purposes of the mainline may be to bind the system interrupts to handling primitives.

Kernel functions such as scheduling can be driven from an interrupt or they can be part of the kernel task itself. Since the main goal of the MDA tool is flexibility, the details of all kernel applications and functions are left entirely to the system designer. The designer can choose to model any type of operating system primitive that he or she can devise.

A.5.3 Task File

All task file names must end with the suffix .task. These files contain the complete specifications of the application tasks in the system. They can include TYPE and/or VAR directives to define structures and data local to the task. In addition, they contain the code for all the activities and procedures of the task as well as the task mainline.

The application software is divided into a number of tasks that can execute concurrently with other tasks. In the MDA system, the form of these application tasks differs greatly from that of the kernel task. The first difference is that unlike the kernel task, application tasks can not have primitives defined within them nor can they define shared data. The second difference is more profound and concerns the actual structure of the application.

2Although variables and types defined in the kernel should logically only be accessible to the kernel, they are actually stored as global variables. Thus they can not have the same names as identifiers defined in the node file. It is good programming practice for the user to code the application tasks so that they do not access data defined in the kernel. However, the MDA tool does not explicitly prevent this occurrence.
Because of the complex and diverse nature of the operations performed by a real-time kernel and because the main kernel task is usually only used for initialization, MDA allows a great deal of freedom in how it can be coded. However, this is not the case for the application tasks. In order to promote a well structured design right from the prototype stage, MDA requires all application tasks to conform to a state machine representation. A typical state machine representation of a task is given in figure A.7.

In this figure, states A, B, and C represent the places at which a task may be suspended due to the actions of system calls. Hence these states represent calls to potentially blocking IPC or synchronization primitives, device handler calls, or any action that may temporarily impede the progress of the task. The transitions between the states represent the individual activities that make up the task. These activities perform the basic functions of the task and once started, they do not require any external data, services, or resources and are therefore atomic entities. In many instances, it is possible to have several transition arcs emanating from a single state. To allow this situation, conditions are placed on the execution of each of these arcs.

This state machine representation imposes an implicit structure on the design of the application tasks. This structure is shown in figure A.8.

As this is a direct translation of any task's state machine representation, it is also the basic control structure chosen for the MDA language. The only modification to the basic structure is that the inner case construct has been replaced by a series of if..endif constructs to allow different conditions to be tested within each state. The syntax of an MDA task file is as follows:

```plaintext
TYPE <Local Type Definition>

TYPE <Local Type Definition>

VAR <Local Variable Definition>

VAR <Local Variable Definition>

PROC <procedure name>
```
Figure A.8: Application Task Structure
<Local Procedure Instructions>
.
ENDPROC;

PROC <procedure name>;
.
<Local Procedure Instructions>
.
ENDPROC;

ACTIVITY <activity name>;
.
<Activity Instructions>
.
ENDACT;

ACTIVITY <activity name>;
.
<Activity Instructions>
.
ENDACT;

BEGIN;
{DO_ACT <activity name>;} 
CALLP <primitive name>, {<parameter list>};
SETSTATE <string constant of state name>;

LOOP;

STATESELECT;

STATE <state name>;
IF <logical expression>;
{DO_ACT <activity name>;} 
CALLP <primitive name>, {<parameter list>};
SETSTATE <string constant of state name>;
ENDIF;
IF <logical expression>;
{DO.ACT <activity name>;} 
CALLP <primitive name>, {<parameter list>};
SETSTATE <string constant of state name>;
ENDIF;
.
.
ENDSTATE;

STATE <state name>;
IF <logical expression>;
{DO.ACT <activity name>;} 
CALLP <primitive name>, {<parameter list>};
SETSTATE <string constant of state name>;
ENDIF;
IF <logical expression>;
{DO.ACT <activity name>;} 
CALLP <primitive name>, {<parameter list>};
SETSTATE <string constant of state name>;

290
CALLP <primitive name>, {<parameter list>};
SETSTATE <string constant of state name>;
ENDIF;
.
.
ENDSTATE;
.
.
ENDSELECT;
ENDLOOP;
ENDTASK;

The TYPE and VAR directives serve the same purpose as in the node and kernel files except that the type and variable definitions are local to the task in which they are defined. That is, they can not be referenced by other tasks. As always, the inclusion of these directives is optional and the VAR directive can be interchanged with the STAT directive.

The variable and type declarations can be optionally followed by the definitions of any number of procedures and activities. There is no limit as to the number of procedures and activities that can be defined. In fact, it is not necessary to define any. However, if both are present, all procedure definitions must precede the activity definitions. The local task procedures are exactly analogous to the procedures of the kernel file. The only difference is that the task procedures may only be called from within the activities defined in the task file.

Activities are very similar to procedures with a few differences. The first of these is that they are delineated by the ACTIVITY and ENDACT directives. These directives serve the same purpose as the PROC and ENDPROC directives described previously. The second difference between procedures and activities is that activities can not have any parameters associated with them. A call to an activity is done using the DO.ACT directive described below rather than with a call instruction (as is the case with procedures).

Following the definition of any procedures and activities is the mainline of the application task. The mainline is started by the BEGIN directive just as in the case of the kernel file. However, from this point, the MDA tool imposes a strict structure on the way the task mainline must be programmed. The BEGIN directive is followed by an initialization state composed of the DO.ACT, CALLP, and SETSTATE directives. These three directives make up the execution sequence of every task state. The DO.ACT directive is similar to the call instruction in that it transfers control to a subroutine (specifically an activity), however it can not have any parameters associated with it. NOTE: In any task state, the DO.ACT directive is optional. If no activity is to be done during a task state, then the DO.ACT directive may be omitted. However, the CALLP and SETSTATE directives must always be present. The CALLP directive is used to call a system primitive defined in the kernel file. It is exactly the same as the call instruction but it is used only to call primitives. In fact, it is the only means by which primitives can be called from outside the kernel file. The CALLP directive may have parameters associated with it. If in some state(s) there is no need to perform any primitive function, then a "nprim" primitive must be defined in the kernel which does nothing but satisfy the requirements for the presence of the CALLP directive in every state. The SETSTATE directive is used to specify which task state is to be executed the next time through the task loop. It must specify the name of the next task state in the form of a string constant.

Following this initialization state, the main loop of the task is defined by the LOOP directive at the top of the loop and by the ENDOLOOP directive at the bottom of the loop. This is an endless loop which is executed for the duration of the simulation. Within the main loop, the STATESSELECT...ENDESELECT directives serve as the case construct for selecting an execution path based on the current task state that was set by the SETSTATE directive executed during the last task cycle.

Inside the STATESSELECT case construct are the definitions for all of the states in the task. Each state is delineated by the STATE...ENDESTATE directives. The STATE directive serves to identify the location of the start of the state while the ENDESTATE directive branches to the bottom of the loop at the end of the state. There can be any number of states in a task. Each state may contain any number of conditional execution clauses, each delineated by the IF...ENDIF directives. These directives are the same as their if and endif instruction counterparts. The only difference
### Event File Syntax

<table>
<thead>
<tr>
<th>Event</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVENT</td>
<td><code>&lt;event name&gt;;</code></td>
<td>{identifier}</td>
</tr>
<tr>
<td>INTERRUPT</td>
<td><code>&lt;interrupt number&gt;;</code></td>
<td>{integer 0-5}</td>
</tr>
<tr>
<td>TYPE</td>
<td><code>&lt;event type&gt;;</code></td>
<td></td>
</tr>
<tr>
<td>PERIOD</td>
<td><code>&lt;event period&gt;;</code></td>
<td>{integer}</td>
</tr>
<tr>
<td>PROBABILITY</td>
<td><code>&lt;event probability&gt;;</code></td>
<td>{integer 0-100}</td>
</tr>
<tr>
<td>QUEUE</td>
<td><code>&lt;max. queue size&gt;;</code></td>
<td>{integer}</td>
</tr>
</tbody>
</table>

```
EVENT       <event name>;       {identifier}
INTERRUPT   <interrupt number>; {integer 0-5}
TYPE        <event type>;      |
PERIOD      <event period>;    {integer}
PROBABILITY <event probability>; {integer 0-100}
QUEUE       <max. queue size>; {integer}
```

Figure A.9: Event File Syntax

is that the IF...ENDIF directives cannot have an else clause. Once a particular state is entered, each IF directive in the state is checked. The body of the first one with a logical expression evaluating to true is executed. **NOTE:** It is imperative that at least one of the IF conditions evaluates to true. For this reason, it is good practice to have each state end with the following IF clause:

```
IF #true;
   ...
ENDIF;
```

This is the equivalent of an “otherwise” clause which is executed if nothing else is true.

Finally, the **ENDTASK** directive is required to mark the end of the application task.

#### A.5.4 Event File

All event file names must end with the suffix `.est`. The event file is used to specify the number and nature of the interrupts associated with each virtual processor. A single event file specifies the interrupts for a single virtual processor. The event file has the format shown in figure A.9.

In order to allow a complete and realistic modeling of environmental events, the MDA system allows for the specification of a large variety of event types. The events associated with each virtual processor are specified by means of the event file which contains a number of directives. There are six different directives and all must be specified for each event. The directives are **EVENT, INTERRUPT, TYPE, PERIOD, PROBABILITY, and QUEUE**. The EVENT directive simply serves to give a symbolic name to the event and the INTERRUPT directive associates a particular interrupt line with the event. The interrupt number must be an integer in the range 0 to 5 corresponding to the valid interrupt numbers.

The TYPE directive specifies whether the event is to be cyclic, sporadic, dependent, or delayed. As such, the directive must include one of these identifiers as the `<event type>`. A cyclic event occurs at regular intervals. For this type of event, the PERIOD directive specifies the number of time units that must elapse between assertions of the event. For cyclic events, the PROBABILITY must always be 100. A sporadic event is the opposite of a cyclic event in that it has no fixed period but it occurs with a certain probability. As such, the PERIOD directive must be 0 while the PROBABILITY directive must be in the range 0 - 100. The determination of whether an event with probability less than 100 will occur during any cycle is based on a random number generator. A
random number between 0 and 100 is generated. If this number is less than the specified probability, then the event will be raised during the cycle.

A dependent event is one that can be raised only through the execution of a cause instruction. A dependent event, when used in conjunction with a task simulating the environment, provides an effective means of modelling environment state dependent occurrences. A dependent event associated with any processor can be raised by a cause instruction on any processor. Thus it is possible to dedicate a virtual processor to modelling the environment while the other virtual processors model the proposed controller. Both the PERIOD and PROBABILITY of a dependent event must be 0.

Finally, a delayed event is one which always occurs some time after another event. For this type of event, the PROBABILITY directive is used to specify the interrupt number associated with the event that the delayed event is to follow. As a result, the probability value must be in the range 0 - 5 (corresponding to the range of interrupts) and it must not be the same as the number specified in the INTERRUPT directive. That is, a delayed event can not follow itself. A delayed event can occur either a fixed amount of time after the triggering event or it can occur a random time interval after the triggering event. In order to specify a fixed lag time, the PERIOD directive should specify a value greater than 0. If the specified PERIOD is 0, then a random lag value of between 1 and 200 time units is selected. A different random value is chosen after each triggering event occurrence.

The final directive used to specify event parameters is the QUEUE directive. This directive is used to specify the number of event occurrences that are to be queued for each event. A queue size of 0 indicates that the event is volatile in nature and if not acted upon immediately, it will be lost. A queue size of 1 indicates that the event is latched. The event occurrence will be saved until it can be acted upon. Larger queue sizes indicate the number of occurrences that are to be saved. For all events with queue sizes greater than 0, the ackint instruction must be used to clear an event occurrence from the queue. If several event occurrences are queued, then several ackint instructions must be executed to clear them.

A.6 File Structure and Loading Sequence

The input files required by MDA follow the hierarchical structure shown in figure A.10. Each of these files consist of various directives along with the actual instructions of the MDA language as described in the previous sections.

The actual loading of the user input files involves reading each line of the input file, compiling it into an internal code format, and then loading it into its proper memory location in the proper virtual processor. The sequence of loading all the source files into the virtual processors closely follows the hierarchical structure of the file system. An example of such a sequence is given in figure A.11. The file loading process starts with the .sys file and then is interrupted by the loading of the first .nod file specified in the .sys file. The loading of the .nod file is then temporarily interrupted by the loading of the .knl file specified in the .nod file. Once the .knl file is loaded, the loading of the .nod
file continues until it is again interrupted, this time by the loading of the .evt file. In other words, loading always starts with the .sys file. As soon as a file name of a lower level file is encountered during the loading of the current file, loading of the current file is temporarily suspended to load the lower level file. Once the lower level file and all of its "child" files have been loaded, the loading of the current file continues.

Of all the input files, only the kernel file and the task files actually generate any executable code. This is because these are the only two files that contain any MDA language instructions. All of the other files consist only of directives that do not result in any compiled code. Each MDA instruction is compiled into an internal format and loaded into the memory space of the appropriate virtual processor. The kernel code is loaded starting at logical address 1000 in the virtual processor. Each instruction occupies one code location. The kernel code can contain up to 1000 instructions.

The application code consists of the instructions of all the application tasks resident in the processor. The code for each task is loaded starting at address 0 and continues until the end of the task code is reached or until address 1000 is reached. That is, each task starts at logical address 0 and can have a maximum of 1000 instructions. The maximum number of tasks per processor is a user modifiable constant. The fact that each application task is loaded starting at logical address 0 does not cause any interference as they are each loaded in different contexts. Again, each instruction occupies one address location.

Because the MDA system requires all identifiers to be defined before they are used, a problem can sometimes arise in the compilation of files that reference shared variables (those defined using the SHARED directive). Shared variables can be referenced by any virtual processor and are defined in any kernel file. Consider the following example. A shared variable var is defined in the kernel of node 2. This variable is referenced by an application task in node 1. The sequence of NODE directives in the .sys file has node 1 declared before node 2. In the loading sequence, the application task referencing var will be loaded before kernel 2 in which the variable is defined. This will cause a loading error in the application task. To avoid such situations, care should be taken to ensure that shared variables have been defined in kernels that will be loaded before any task (on any virtual processor) using those variables is loaded. This can be ensured by defining all shared variables in the kernel of the first node to be loaded.

A.6.1 File Structure Usage

The main motivations behind the MDA file system are encapsulation of functionality and software reusability. The divisions between the five MDA input file types allow the system designer to independently design application software, kernel software, define event characteristics, and configure the system in terms of the number of processors. Because each of these specifications and configurations is contained within one or more discrete files, it is very simple to experiment with different setups.
For example, task to processor allocation is accomplished simply by moving the TASK directives between .nod files. Similarly, different kernels can be tried in different processors by simply changing the KERNEL directive in the appropriate .nod file(s). These are only two examples but the same reasoning applies to all configurable aspects of the system.

Another benefit of the file system structure is that the same individual tasks, kernels, or even complete nodes can be used in different places in the system by simply copying directives. In this manner, the same task or kernel can be resident on each processor, for example. Also, standard tasks and kernels can be saved in their appropriate .tsk and .knl files for use when needed.

As a consequence of its file structure, MDA allows different user defined kernels to run on different virtual processors during a simulation. In this way, the interactions of different types of kernel functions can be directly observed. This flexibility in kernel design does not mean that a system designer always has to program a kernel if it is only the application software behavior that needs to be analyzed. It is possible to define standard primitives such as the semaphore signal and wait or the standard Harmony primitives send, receive, and reply and keep them in a permanent kernel file. This standard kernel, once defined, can then be used as though it is a built-in part of the MDA tool as far as the application software is concerned.

### A.7 MDA Output Files

In addition to the MDA input files, there are two types of output files produced by the MDA system. The output files are the listing files and the statistics file.

#### A.7.1 Listing File

As each of five input files is compiled and loaded, the MDA tool creates a listing file for each loaded file. For example consider a system consisting of two processors specified by the files buf.sys, first.nod, second.nod, clk evt, sema1.knl, sema2.knl, task1.tsk, and task2.tsk. The hierarchical structure of this file system is given in figure A.12. Note that the same tasks are resident on both processor 0 and on processor 1. The following listing files would be produced by the MDA tool in this case:

- buf0.lis, first0.lis, clk0.lis, sema10.lis, task10.lis, task20.lis, second1.lis, clk1.lis, sema21.lis, task11.lis, and task21.lis. It should be noted that a .lis file is created for every input file. If the same file is loaded into more than one processor, then more than one .lis file is produced for it. The name of the .lis file is made up of the input file name, followed by a number indicating the processor on which it was loaded, followed by .lis.

A listing file contains a copy of the input source file. Each statement in the source code is sequentially numbered and accompanied by the address at which it was loaded in the given virtual processor. Statements that do not generate any compiled code (such as most directives) do not have
loading address locations associated with them. The load address locations are generated as each file is loaded. The listing file is extremely useful in loading and running MDA simulations. In an error occurs during the loading of a file, the MDA loader provided the sequential number of the offending instruction. This information can be cross referenced with the listing file. Similarly at run time, if an error occurs, the MDA interpreter provides the address of the instruction that caused the error. Again this information can be cross referenced with the listing file to find the cause of the problem.

A.7.2 Statistics File
At the start of a simulation run, the MDA system asks the user to enter the name of an output statistics file. This file is then used as the output for any *fprintf*, *ftype*, and *ftypecr* instructions executed by the virtual processors. In addition, all of the built-in and user defined statistics collected by the prototype are copied to this statistics file during and at the end of each simulation.

A.8 Starting and Ending Simulations
Once all of the input files required by the MDA tool have been completed, the actual simulation can begin. The MDA simulator is invoked by typing *sim* in the directory on which the executable simulator file is resident. This action results in the following prompt:

```
REAL-TIME SYSTEM SIMULATOR
```

Enter System Descriptor File Name >>

This prompt should be responded to by typing the complete name of the system descriptor input file. Once this is done, the following prompt occurs:

Enter Statistics File Name >>

This prompt should be responded to by entering the name of the file to which the user wants the statistical information printed. If a file of the specified name already exists, it will be overwritten. All of the input files required by the MDA system must be resident in the directory from which the simulator has been invoked. Once the initial prompts have been satisfied, loading commences. The loading process generates status lines like the following:

```
Loading System Descriptor File : buf.sys
Loading Node Descriptor File   : NODE1.nod
Loading Kernel Descriptor File : SEMA.knl
Loading Event Descriptor File  : CLK.evt
Loading Task Descriptor File  : PROD1.tsk
Loading Task Descriptor File  : PROD2.tsk
Loading Task Descriptor File  : CONS.tsk
Loading Node Descriptor File   : NODE2.nod
Loading Kernel Descriptor File : SEMA.B.knl
Loading Event Descriptor File  : CLK2.evt
Loading Task Descriptor File  : PROD1.tsk
Loading Task Descriptor File  : PROD2.tsk
Loading Task Descriptor File  : CONS.tsk
```

ALL FILES SUCCESSFULLY LOADED - SYSTEM CONFIGURATION:

<table>
<thead>
<tr>
<th>NODE</th>
<th>NODE1</th>
<th>NODE2</th>
</tr>
</thead>
<tbody>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
</tbody>
</table>

296
Enter Simulation Time (0 to Abort)  >>

This output serves to show the progress of loading. Once loading has been completed, the system configuration is shown. Finally a prompt is displayed asking for the length of the simulation run in units of time. The maximum value to this response depends on the data sizes used by the host machine but will never be less than 2,147,483,647. Hopefully this will be enough for any purpose. If the user wants to abort the simulation immediately after loading, a simulation time of 0 can be entered.

At the end of the simulation, a message is displayed indicating that the simulation has successfully completed. In addition to this, a prompt is issued asking if the simulation time is to be increased. If this is the case, the user enters the additional amount of time required. If no extra time is needed, then the user enters 0 to complete the simulation.

## A.9 Run Time Debug Environment

The MDA tool provides a run time debug environment that can be used to monitor and modify the execution of a running simulation. The debug environment is entered when the `brpt` instruction is executed during the course of a simulation or when the <CR> key is typed during the course of a simulation. When either of these events occur, the following information is printed to the terminal:

**Current Execution Status:**

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed Execution Time</td>
<td>12345</td>
</tr>
<tr>
<td>Requested Execution Time</td>
<td>7777727</td>
</tr>
<tr>
<td>Currently Executing in Node</td>
<td>NODE1</td>
</tr>
<tr>
<td>Virtual Processor Time</td>
<td>2274</td>
</tr>
<tr>
<td>Currently Executing in Task</td>
<td>PROD2</td>
</tr>
<tr>
<td>Last instruction executed at code location</td>
<td>46</td>
</tr>
<tr>
<td>Last Instruction Executed</td>
<td>MOVE</td>
</tr>
<tr>
<td>Current PC Value</td>
<td>47</td>
</tr>
</tbody>
</table>

**V(alue), S(tep, G(roup step, E(vents, C(ontinue, Q(uit >>

The *elapsed execution time* gives the amount of simulated time that has elapsed in the simulation. The *requested execution time* gives the total requested length of the simulation. The *currently executing in node* field specifies the name of the virtual processor on which the last instruction was executed. *Virtual processor time* gives the time elapsed on the virtual processor specified in the "currently executing in node" field. *Currently executing in task* gives the name of the task which is the current context on the name virtual processor. The last three fields are self explanatory.

Following the information dump is a menu line that specifies the options available to the user and waits for a reply to the prompt. Each of the options is invoked by typing the first letter of the option. The `step` option allows single stepping through the program instructions one at a time. After each instruction is executed, the user is returned to the debug environment. The `group step` option is similar except that it executes a number of instructions before returning to the debug environment. The number of instructions is specified by the user in response to a later prompt. The `continue` option simply resumes execution from the point of interruption. The `quit` option causes the simulation to terminate.

297
The values option allows the user to examine the contents of any type of variable declared in any scope of the modeled system. Once this option is chosen, the debugger prompts for the scope of the required variable. This can be any one of L(ocal), G lobal, or S(hared). The user must respond with L, G, or S to this prompt. If the variable is local or global, another prompt is issued asking for the name of the virtual node (processor) in which the variable is defined. If it is a local variable, a third prompt is issued asking for the name of the task in which it is defined. Finally, for all variable scopes, a prompt is issued asking for the name of the variable. The variable name entered can be qualified if the variable is of a user defined type. If the variable is found, its current value is printed to the screen along with its type and its size. In the case of user defined types, a hexadecimal dump of the structure contents is given allowing the user to find the individual fields within the structure. (Specific base type fields can be printed normally by qualifying the structure name). In the case of stacks and queues, each element of the stack or queue is printed along with its position (and priority in the case of queues). After the option has been completed, the user is returned to the main debug menu.

The events option allows the user to observe and modify the data associated with any event defined in the system. When this option is invoked, it prompts for a virtual processor name and an interrupt number. After these items have been entered by the user, all of the characteristics of the interrupt (event) are displayed including:

1. Event type,
2. Name,
3. Handler location,
4. Period (for cyclic events),
5. Current count (for periodic events),
6. Maximum queue size,
7. Number of such event occurrences currently queued,
8. The interrupt's active state,
9. Probability (for sporadic events),
10. The interrupt that is to be followed (for delayed events),
11. The delay period (for delayed events)

After these values are displayed, the user is free to change them using the events menu. Once the events option is complete, the user is returned to the main debug menu.
A.10 MDA Statistics Facilities

In recognition of the fact that prototypes built using the MDA tool will need to be evaluated and compared to other such prototypes, the MDA system contains a number of builtin statistics that are automatically gathered and printed to the statistics file at the end of the simulation. In addition, MDA provides facilities for the user to program his or her own statistics gathering functions as part of the kernel or task code using standard MDA instructions.

A.10.1 Built-in Statistics

Data belonging to the built-in class is automatically collected by the MDA interpreter and is fixed for all application prototypes. Several such statistical items are gathered by the MDA interpreter during the course of the execution:

1. **Number of Virtual Processors.** This data is simply recorded after all files have been loaded.

2. **Total Memory Conflicts.** This item represents the number of times that two or more processors accessed shared data variables (either for a read or a write type operation) during the same cycle of the MDA execution engine. This data is also broken down by shared data variable. That is, data is collected on the number of access conflicts on a per variable basis.

3. **Total Shared Memory Writes.** This item represents the number of times each shared memory variable was overwritten. It is collected on a per variable basis.

4. **Final CPU Time.** This data gives the final elapsed CPU time for each virtual processor.

5. **Context Switches.** This data shows how many context switches took place in each virtual processor during the course of the simulation.

6. **Task Data.** The number and identities of all tasks is also recorded on a per processor basis after all of the MDA files have been loaded.

7. **Maximum and Average Queue Lengths.** For every queue or stack variable defined by the user in the prototype and for each event queue defined to be of size greater than 1, the MDA interpreter keeps track of the maximum number of elements contained in the stack or queue (list) over the course of the simulation and the average number of elements held in the list. This average is determined by calculating a weighted length:

\[
\text{maxlen} = \sum_{\text{length} \geq 0} \text{length} \times \text{time} \times \text{length}
\]

in which \text{length} is the length of the list and \text{time} is the elapsed time for which the list had a size of \text{length}. The average list length is then be calculated by dividing the weighted length factor by the total elapsed virtual CPU time.

At the end of the simulation, all of these statistics are automatically formatted and copied to the statistics file.

A.10.2 User Collection of Statistics

In realization of the fact that it is impossible to predict all the types of data that may be needed for the complete analysis of a particular application, MDA allows users to program the collection of their own statistics using the same environment and instructions used to program the system model.

Every MDA instruction that is not part of the prototype but is programmed only for user data collection should be preceded by an asterisk (*). This signals to the interpreter that the corresponding instruction is to be executed but it is not to take any virtual CPU time. Further, to prevent problems in interprocessor instruction execution timing, all such consecutive instructions on any processor are executed contiguously without the normal interleaving through each virtual
processor after each instruction. Thus each block of data gathering instructions is executed as an
atomic entity that takes no time to run.

The second item that needs to be addressed with user programmed statistics gathering is how
the data is to be communicated to the outside world. The data collected by the prototype is stored
in the data structures declared within the model through the VAR and SHARED directives. In
addition to these, there is a STAT directive. This directive is used in exactly the same manner as
the VAR directive. A STAT variable can be used for any purpose but at the end of a simulation
run, MDA goes through all STAT variables and copies their names and values to a user specified
output statistics file. If the STAT variable is of a user defined type, the value of each field is copied
to the statistics file. An example of code for a statistics gathering function along with its results is
shown in figure A.13.

A.10.3 Simulated Time

Many of the user defined statistics used in MDA simulations need some type of time basis. In the
MDA environment, each instruction execution requires 1 unit of regardless of the type of instruction.
As such, each instruction execution increases the virtual processor time by one unit. The only
exception to this situation arises when an instruction is preceded by an asterisk. The execution of
these instructions does not require any virtual cpu time. In addition to instruction execution, virtual
cpu time is incremented by any specified value using the *ticks instruction. This instruction is
used to simulate actions that require some amount of time. To facilitate user statistics gathering,
timestamps are taken using the time instruction which returns the current virtual cpu time.

A.11 MDA System Configurability

The MDA tool has been implemented in both UNIX/C and Turbo C. As such the tool can be
run both on DOS based machines as well as UNIX based machines. The two implementations are
very similar except that due to memory restrictions on the DOS based machines, the Turbo C
implementation is useful for only small simulations using one or two virtual processors with three
or four tasks each. In order to allow fast simulation times, all of the compiled code for each virtual
processor is resident in the main memory of the host machine at all times during the simulation.
Thus the number of tasks and virtual processors is limited only by the memory capacity of the host
machine.

The MDA source code comes as a series of C language files, each of which can be separately
compiled. One of these files utiltypes.h, contains all of the type declarations global to all of the MDA
C source files. Included in this file are a number of C #define directives that specify the constant
values to be used in determining the size of the simulations that can be supported by the MDA
tool. To configure the tool to run larger simulations or to allow the tool to run on host machines
with limited memory, these constant values can be changed and the system recompiled to produce
a more suitable object file. These constants are defined in the following list.

1. MAX_NODES - This constant defines the maximum number of virtual processors that can
be configured. (Standard Value: 5).

2. MAX_TASKS - This constant defines the maximum number of tasks that can be defined
per virtual processor. (Standard Value: 20).

3. MAX_INTS - This constant defines the maximum number of interrupts per virtual proces-
sor. (Standard Value: 6).

4. MAX_INTQ - This constant defines the maximum size to which an event queue can be
configured. (Standard Value: 10).

5. MAXCONSTS - This item defines the maximum number of constants that can be expected
in the input files. (Standard Value: 300).

6. MAX_SHARED - This item defines the maximum number of structures or variables that
can be defined in shared memory. (Standard Value: 50).
; PROCEDURE sched_rtr_task (rtr_tcb : t_tcb)
;
; This procedure accepts a process control block and adds it to the ready to run queue in a prioritized fashion.

PROC sched_rtr_task;
   *time start_sched;
   pop taskstk, rtr_tcb;
   move #/ready/, rtr_tcb.status;
   addq rtr_tcb, rtr, rtr_tcb.priority;

   if (rtr_tcb.priority > curr_tcb.priority);
      pop taskstk, ret_pc;
      pop taskstk, ret_pc
      move ret_pc, curr_tcb.pcnt;
      move taskstk, curr_tcb.stk;

      move #/ready/, curr_tcb.status;
      addq curr_tcb, rtr, curr_tcb.priority;

      *time end_sched;
      *sub end_sched, start_sched, elapsed;
      *ftype #/Time Spent in Scheduler: /;
      *ftypecr elapsed;

   call run_rtr_task;
   endif;

   *time end_sched;
   *sub end_sched, start_sched, elapsed;
   *ftype #/Time Spent in Scheduler: /;
   *ftypecr elapsed;
ENDPROC;

Output Produced in Statistics File:

Time Spent in Scheduler: 24

Figure A.13: User Statistics Gathering
7. **MAX. QUEUE** - This constant defines the number of queues or stacks for which builtin statistics will be kept. More than this number of queues and stacks can be defined and used but builtin statistics will not be kept for them. (Standard Value: 50).

8. **MAX. EV. Q** - This constant defines the number of event queues for which builtin statistics will be kept. More than this number of event queues can be defined and used but builtin statistics will not be kept for them. (Standard Value: 15).

All of the standard values provided in the above list are for the UNIX/C implementation of MDA running on Digital Equipment Corporation RISCstations.

### A.12 Complete MDA System Example

The following example shows the contents of all of the files that would be needed by MDA to model a simple producer consumer problem using a single virtual processor. The producer tasks PROD1 and PROD2 send data items to a consumer task CONS. After they have done so, they change their task priorities. The data exchange is done using a semaphore based kernel.

#### A.12.1 The System Descriptor File

The system descriptor file *buf.sys* would contain the following:

```plaintext
; Producer Consumer Problem Using Semaphores
;
NODE node1;
```

#### A.12.2 The Node File (node1.nod)

```plaintext
; NODE1 FILE
KERNEL sema;
TYPE t_sema = count : integer; semaphore counter
  wait_q : queue; semaphore wait queue
VAR buf_avail, t_sema; semaphores used by application tasks
VAR mutex, t_sema;
VAR item_avail, t_sema;
EVENT clk; only external event is clock interrupt
TASK prod1, 50;
TASK prod2, 50;
TASK cons, 100;
ENDNODE;
```

#### A.12.3 The Event File (clk.evt)

```plaintext
event clock;
interrupt 0;
type cyclic;
period 7;
probability 100;
queue 1; latched event
```

#### A.12.4 The Kernel File (sema.knl)

```plaintext
; KERNEL CODE FOR SEMAPHORE OPERATIONS
;
KERNEL sema;
TASK kern_task, 50;
```

302
TYPE t_pcb = name : string *; Process Control Block - task name
  status : string *; task status
  prio : integer *; task priority
  pcnt : integer *; task program counter
  stk : stack; task stack

TYPE t_buffer = size:integer, buf_item:queue; Buffer Type

VAR args, stack;
VAR tlist, stack;
VAR semptr, string;
VAR semcnt, integer;
VAR w_pc, integer;
VAR no_prempt, boolean;
VAR ret_pc, integer;
VAR t_wait_q, queue;
VAR temp_pcb, t_pcb;
VAR pcb_name, string;
VAR sp_prio, string;
VAR br_chk, integer;
VAR cnt, integer;
VAR nil_queue, queue;
VAR error, boolean;
VAR new_pc, integer;
VAR sc_args, stack;
VAR sc_tlist, stack;
VAR rtr_pcb, t_pcb;
VAR rtr_prio, integer;
VAR cur_prio, integer;
VAR parms, stack;
VAR template, stack;
VAR tempstr, string;
VAR t_proclist, queue;
VAR cur_pcb, t_pcb;
VAR rtr, queue;
VAR buf_list, t_buffer;
VAR buffer, queue;

; This procedure picks the next task from the ready to run queue
; for execution.
PROC run_rtr_task;
  case rtr, br_chk;
    if (br_chk = 0)
      printer "ERROR: NO RTR PROCESSES/";
      halt;
    else
      delq rtr, cur_pcb;
      move #/running/, cur_pcb.status;
      move cur_pcb.name, pcb_name;
      extsw cur_pcb.name;
      move cur_pcb.stk, taskstk;
      enable 0;
      move cur_pcb.pcnt, pc;
    endif;
  ENDPROC;

; This procedure adds a task to the ready to run queue according to
; its priority.  PARAMETERS: Gpcb : string;
PROC sched_rtr_task;
  pop taskstk, rtr_pcb;
  move #/ready/, rtr_pcb.status;
  addq rtr_pcb, rtr_rtr_pcb.prio;
  if (rtr_pcb.prio > cur_pcb.prio);
    pop taskstk, ret_pc;
    move ret_pc, cur_pcb.pcnt;
    move taskstk, cur_pcb.stk;
    move #/ready/, cur_pcb.status;
    addq cur_pcb, rtr, cur_pcb.prio;
    call run_rtr_task;
    printer "ERROR: SCHEDULE TASK/";
    halt;
  endif;

303
This is the clock interrupt handling primitive.

PRIM clk_hnd;

disable $0;
ackint $0;
printer/"clock interrupt occurred./

pop taskstk, w_pc;
move w_pc, cur_pcb.pcnt;
move taskstk, cur_pcb.stk;
move $/ready/, cur_pcb.status;
addq cur_pcb, rtr, cur_pcb.prio;
call run_rtr_task;
ENDPRIM;

PRIM nilprim;
ENDPRIM;

This primitive allows the calling task to set its own priority.
PARAMETERS: sp_prio : integer;

PRIM setprio;

disable $0;
pop taskstk, sp_prio;
move $sp_prio, cur_pcb.prio;
siseq rtr, br chk;

; Preempt task if it no longer of highest priority
if (br chk > $0);
move rtr, t_wait_q;
delq t_wait_q, temp pcb;
if (temp pcb.prio > cur_pcb.prio);
pop taskstk, ret_pc;
move ret_pc, cur_pcb.pcnt;
move taskstk, cur_pcb.stk;
move $/ready/, cur_pcb.status;
addq cur_pcb, rtr, cur_pcb.prio;
call run_rtr_task;
printer $/ERROR: SETPRI/;
halt;
endif;
ENDPRIM;

This procedure allows a calling task to get its current priority
PARAMETERS: sp_prio : string; (pointer to location where priority
value is to be placed)

PRIM getprio;

disable $0;
pop taskstk, sp_prio;
move cur_pcb.prio, $sp_prio;
enable $0;
ENDPRIM;

The following two procedures allow the calling task to add items to
the buffer queue and to remove items from the buffer queue.

PRIM putbuf;
pop taskstk, tempstr;
addq @tempstr, buffer, $100;
ENDPRIM;

PRIM getbuf;
pop taskstk, tempstr;
delq buffer, @tempstr;
ENDPRIM;

This is the initialize semaphore primitive.
PRIM initm;
    disable $0;
    pop taskstk, cnt;
    pop taskstk, semptr;
    move cnt, @semprt.count;
    move nil_queue, @semprt.wait_q;
    enable $0;
ENDPRIM;

; This is the semaphore WAIT primitive
PRIM waitm;
    disable $0;
    pop taskstk, semptr;
    if ($semprt.count = 0);
        move $/wait/, cur_pcb.status;
        pop taskstk, w_pc; Pop Return Address From Stack
        move taskstk, cur_pcb.stk;
        move w_pc, cur_pcb.pcnt;
        addq cur_pcb, @semprt.wait_q, $100;
        call run_rtr_task;
        printc $/ERROR - WAITING/;
        halt;
    else;
        sub @semprt.count,$1, @semprt.count;
    endif;
    enable $0;
ENDPRIM;

; This is the semaphore SIGNAL primitive
PRIM sigam:
    disable $0;
    pop taskstk, semptr;
    sizeq @semprt.wait_q, br_chk;
    if (br_chk = 0);
        add $1, @semprt.count, @semprt.count;
    else;
        delq @semprt.wait_q, temp_pcb;
        call sched_rtr_task, temp_pcb;
    endif;
    enable $0;
ENDPRIM;

BEGIN: { Task Mainline }
    disable $0;
    move tasklist, t_proclist;
    sizeq t_proclist, br_chk;
    move curtask, tempstr;
    while br_chk <> 0;
        delq t_proclist, pcb_name;
        cxtsw pcb_name;
        move taskstk, temp_pcb.pcnt;
        move pcb_name, temp_pcb.pcnt;
        move $/ready/, temp_pcb.status;
        move ccpprio, temp_pcb.prio;
        move taskstk, temp_pcb.stk;
        addq temp_pcb, rtr, ccpprio;
        sizeq t_proclist, br_chk;
    endwhile;
    cxtsw tempstr;
    call initm, $/mutex/, $1;
    call initm, $/buf_avail/, $2;
    call initm, $/item_avail/, $0;
    bind clk_hnd, $0;
    call run_rtr_task;
    printc $/ERROR - KERNEL TERMINATING/;
END TASK;
ENDKERNEL;
A.12.5  The Task Files

This section contains the code for the three application tasks: PROD1, PROD2, and CONS. Each is presented in its own subsection.

A.12.5.1 The prod1.tsk File

```
; PRODUCER TASK 1 CODE

VAR count, integer;
VAR myprior, integer;
VAR delta, integer;

ACTIVITY init:
  move $10, delta;
  move $0, counter;
ENDACT;

ACTIVITY startact:
  add counter, $1, counter;
  print $/Produced Message - /
  printer counter;
ENDACT;

ACTIVITY present:
  print currtask;
  print $/Sent Message - /
  printer counter;
ENDACT;

ACTIVITY updpric:
  if (mypirow >= $70):
    move $-10, delta;
  else:
    move $10, delta;
  endif;
  add myprior, delta, myprior;
ENDACT;

ACTIVITY prtpri:
  print currtask;
  print $/Priority = /
  printer myprior;
ENDACT;

BEGIN
  DO.ACT init;
  CALLP nilpri;
  SETSTATE $/start/;

  LOOP:
    STATESSELECT:

    STATE start:
      IF $true;
        DO.ACT
        CALLP
        SETSTATE
        print $/buf avail/
        SETSTATE
        $/wait/
        ENDIF;
        ENDSTATE;

    STATE wait:
      IF $true;
        CALLP
        SETSTATE
        $/mutex/
        END;
        ENDSTATE;

    STATE store:
      IF $true;
        CALLP
        SETSTATE
        $/getprio/
        END;
        ENDSTATE;

    STATE getprio;

706
```
A.12.5.2 The prod2.tsk File

<table>
<thead>
<tr>
<th>PRODUCER TASK 2 CODE</th>
</tr>
</thead>
</table>

VAR counter, integer;
VAR myprior, integer;
VAR delta, integer;

ACTIVITY init:
  move #10, delta;
  move #1000, counter;
ENDACT;

ACTIVITY startact:
  add counter, #1, counter;
  print counter, #/Produced Message -/;
  print counter, #/time message -/;
ENDACT;

ACTIVITY present:
  print currtask;
  print myprior, #/Priority =/;
  print counter;
ENDACT;

ACTIVITY updprio:
  if (myprior < #70);
    move #10, delta;
  else;
    move #10, delta;
  endif;
  add myprior, delta, myprior;
ENDACT;

ACTIVITY prtpri;
  print currtask;
  print myprior, #/Priority =/;
  print counter;
ENDACT;

BEGIN
  DO.ACT init;
  CALLP nilprim;
  SETSTATE #/start/;

ENDTASK;
LOOP;
STATE SELECT;
STATE start;
   IF Status;
      DO_Act;
      CALLP
      SETSTATE
      ENDIF;
   ENDSTATE;
STATE wait;
   IF Status;
      DO_Act;
      CALLP
      SETSTATE
      ENDIF;
   ENDSTATE;
STATE store;
   IF Status;
      DO_Act;
      CALLP
      SETSTATE
      ENDIF;
   ENDSTATE;
STATE getprio;
   IF Status;
      DO_Act;
      CALLP
      SETSTATE
      ENDIF;
   ENDSTATE;
STATE adjprio;
   IF Status;
      DO_Act;
      CALLP
      SETSTATE
      ENDIF;
   ENDSTATE;
STATE prtprio;
   IF Status;
      DO_Act;
      CALLP
      SETSTATE
      ENDIF;
   ENDSTATE;
STATE utxavail;
   IF Status;
      DO_Act;
      CALLP
      SETSTATE
      ENDIF;
   ENDSTATE;
END LOOP
END TASK

A.12.5.3 The cons.tsk File

; CONSUMER TASK CODE
;
VAR counter, integer;
VAR myprio, integer;
VAR dummy, integer;
VAR limit, integer;
VAR task, stack;

ACTIVITY init;
   print curtask;
   print "# Waiting to Receive Message.
ENDACT;

ACTIVITY prmsg;
   print curtask;

308
print $/ Received Message = /;
print counter;
ENDACT;
ACTIVITY updrio;
sub myprior, $20, myprior;
ENDACT;
ACTIVITY pprio;
print current;
print $/ Priority = /;
print myprior;
ENDACT;
BEGIN
DO_ACT nilact;
callp nilprim;
setstate $/start/;
loop;
stateSelect;
STATE start;
if $true;
do_act
callp
setstate
endif;
endstate;
STATE wmutex;
if $true;
do_act
callp
setstate
endif;
endstate;
STATE gbuf;
if $true;
do_act
setstate
endif;
endstate;
STATE deprio;
if $true;
do_act
setstate
endif;
endstate;
STATE adjprio;
if myprior > $40;
do_act
setstate
endif;
if myprior <= $40;
callp
setstate
endif;
endstate;
STATE pprio;
if $true;
do_act
setstate
endif;
endstate;
STATE wbuf;
if $true;
do_act
setstate
endif;
endstate;
endsel;
Appendix B

Prototype Source Code

This appendix contains the Design-Aid language source files that were used to model the systems described in the thesis. Because there was a large number of prototypes developed, the source code has been divided into sections and is presented in the same order as it was encountered in chapter 6.

B.1 Application Software Partitioning

This section contains code for the software partitioning process. The node descriptor file, kernel, and IDLE task code were used in all examples of this section. Their code is presented below.

B.1.1 Node File

```plaintext
; NODE_FILE

; t_trace_elt act values:
#1 - Msg Sent
#2 - Msg Accepted
#3 - Shared Data Write
#4 - Shared Data Read
#5 - Response Out
#6 - Event Trigger
#7 - Start Read Delimiter
#8 - End Read Delimiter
#9 - Start Msg Delimiter
#10 - End Msg Delimiter

; GLOBAL TYPE DEFINITIONS

TYPE t_trace_elt = node_name:string *;
task_name:string *;
act : integer *;
time : integer *;
misc1 : integer *;
misc2 : integer *;

ACTION to be done

TIME of occurrence

TYPE t_msg_elt = data : integer *;
to : integer *;

VAR node_name, string ; Symbolic node name

EVENT clk;

KERNEL baseb;

TASK t1, 51;
TASK t2, 50;
TASK t3, 50;
```
B.1.2 Kernel Code

; KERNEL CODE FILE

; KERNEL base

TASK kern_task, 80;

TYPE t_tcb_state =
  rdy_time : integer *; Total ready time
  run_time : integer *; Total run time
  wt_time : integer *; Total wait time
  start : integer *;
  end : integer *;
  resp_q : queue *; of t_trace_elt

TYPE t_shared_state =
  data_trace ; queue *; of t_trace_elt

TYPE t_tcb =
  name : string *; task name
  num : integer *;
  status : string *; task state
  priority : integer *; priority
  pont : integer *; program counter
  m_wait : string *; queue waited upon
  elt_str : string *;
  msg_in : integer *; signal for msg rx
  stk : stack *; local task stack
  sta : t_tcb_state *; task control block

TYPE t_q =
  list : queue *;
  trace : queue *; list of trace elements of msg elements
  wait : queue *; list of tcb's waiting for queue element

; Message queues

VAR q1, t_q1;
VAR q2, t_q2;
VAR q3, t_q3;
VAR q4, t_q4;
VAR q5, t_q5;
VAR q6, t_q6;
VAR q7, t_q7;
VAR q8, t_q8;
VAR q9, t_q9;
VAR q10, t_q10;
VAR q11, t_q11;

STAT sends, integer;
STAT os_oshd, integer;
VAR start_os, integer;
VAR end_os, integer;
VAR rdy_str, string;
VAR wt_str, string;
VAR run_str, string;
VAR end_str, string;
VAR swait, queue;
VAR ut_chk, integer;
VAR meg_str, string;
VAR dest_str, string;
VAR temp_tcb, t_tcb;
VAR temp_tcb, t_tcb;
VAR ret_pc, integer;
VAR tmp_stk, stack;
VAR got_msg, integer;
VAR num_parms, integer;

; Global Kernel Variables

VAR curr_tcb, t_tcb ; current task control block
VAR proclist, queue ; copy of task name list
VAR br_chk, integer ; branch condition check
VAR tempstr, string ; temporary string value
VAR tcb_name, string ; current task name
VAR xtr, queue ; ready to run queue

; Variables for response_update procedure
VAR action, integer; action requested in response_update
VAR response_alt, t_trace_elt; new response trace element
VAR msg_elt_name, string; name of msg elt for response trace
VAR dest_elt_name, string; name of msg queue
VAR delim_elt, msg or data trace delimiter
VAR q_size, integer; size of trace queue
VAR data_name, string; name of shared data variable
VAR temp_q, queue; data response queue copy

; VARIABLE declarations for procedure NUM_KTA_TASK
VAR error, boolean; Set if no ready processes

; VARIABLE declarations for procedure SCHED_KTA_TASK
VAR rtx_tcb, t_tcb; PCB to be scheduled

; PROCEDURE response_update (Name : string;
; action: integer)
; This procedure updates the response trace queues
PROC response_update:
*pop taskstk, action;
*move nod_name, trace_elt.nods_name;
*move curr_tcb, trace_elt.task_name;
*move trace_elt.time;
*move action, trace_elt.act;
  *if (action = $1); SEND MSG
  *pop taskstk, dest_elt_name;
  *pop taskstk, msg_elt_name;
  *addq trace_elt, curr_tcb.sts.resp_q, $1;
  *move curr_tcb.sts.resp_q, temp_q;
  *sizeq temp_q, q_size;
  *move q_size, $msg_elt_name, trg_size;
  *while (q_size > 0);
  *delq temp_q, trace_elt;
  *addq trace_elt, $dest_elt_name, trace, $1;
  *sizeq temp_q, q_size;
  *endwhile;
  *endif;
  *if (action = $2); RCV MSG
  *pop taskstk, dest_elt_name;
  *pop taskstk, msg_elt_name;
  *move trace_elt, delim_elt;
  *addq trace_elt, curr_tcb.sts.resp_q, $1;
  *move $9, delim_elt.act;
  *addq delim_elt, curr_tcb.sts.resp_q, $1;
  *move $msg_elt_name, trg_size, q_size;
  *while (q_size > 0);
  *delq $dest_elt_name, trace, trace_elt;
  *addq trace_elt, curr_tcb.sts.resp_q, $1;
  *sub q_size, $1, q_size;
  *endwhile;
  *move $10, delim_elt.act;
  *addq delim_elt, curr_tcb.sts.resp_q, $1;
  *endif;
  *if (action = $3); DATA WRITE
  *pop taskstk, data_name;
  *addq trace_elt, curr_tcb.sts.resp_q, $1;
  *move curr_tcb.sts.resp_q, @data_name.stats.trace_q;
  *endif;
  *if (action = $4); DATA READ
  *pop taskstk, data_name;
  *move trace_elt, delim_elt;
  *addq trace_elt, curr_tcb.sts.resp_q, $1;
  *move $7, delim_elt.act;
  *addq delim_elt, curr_tcb.sts.resp_q, $1;
  *move @data_name.stats.data.trace, temp_q;
  *sizeq temp_q, q_size;
  *while (q_size > 0);
  *delq temp_q, trace_elt;
  *addq trace_elt, curr_tcb.sts.resp_q, $1;

313
PROCEDURE run_rtr_task

This procedure takes the first ready to run task off the rtr_queue (this is the one with the highest priority) and copies its saved state into the current processor state. If there are no ready to run tasks, then an error message is printed and the kernel terminates.

PROC run_rtr_task:
  sizeq rtr, br_chk;
  IF (br_chk = 0):
    printer $/ERROR: NO RTR PROCESSES/;
    halt;
  ELSE:
    delq rtr, curr_tcb;
    move curr_tcb.name, tcb_name;
    move curr_tcb.tag, tag;
    move curr_tcb.mag.in, got_mag;
    *time curr_tcb.sta.end;
    add curr_tcb.sta.start, curr_tcb.sta.start;
    add curr_tcb.sta.time, curr_tcb.sta.time;
    move curr_tcb.sta.time, $/rtr_tcb/
    move curr_tcb.pent, po;
    ENDIF;
  ENDPROC;

PROCEDURE sched_rtr_task (t_tcb : t_tcb)

This procedure accepts a process control block and adds it to the ready to run queue in a prioritized fashion.

PROC sched_rtr_task:
  pop taskstk, rtr_tcb;
  move */ready/, rtr_tcb.status;
  *time rtr_tcb.end;
  *add rtr_tcb.sta.end, rtr_tcb.sta.start;
  add rtr_tcb.sta.time, rtr_tcb.sta.time;
  move taskstk, tmp_stk;
  *time rtr_tcb.start;
  addq rtr_tcb, rtr_tcb.priority;
  if (rtr_tcb.priority > curr_tcb.priority):

314
pop taskstk, ret_pc;
pop taskstk, ret_pc
move ret_pc, curr_tcb.pcnt;
move taskstk, curr_tcb.stk;

*time curr_tcb.ste; end;
*shb curr_tcb.ste, end, curr_tcb.ste.start, curr_tcb.ste.start;
*add curr_tcb.ste.start, curr_tcb.ste.run_time, curr_tcb.ste.run_time;
*move curr_tcb.ste.run_time, $run_str;
*time curr_tcb.ste.start;

move #/ready, curr_tcb.status;
add curr_tcb, str, curr_tcb.priority;
call run_rte_task;
halt;
end;
ENDPROC;

; PRIMITIVE nilprim
;
; This primitive performs a dummy operation
PAIN nilprim;
ENDPAIN;

; PRIMITIVE send
;
; This primitive performs a nonblocking send operation
PAIN send;

disable #0;
*add $1, sends, sends;
*time start_os;
pop taskstk, msg_str;
pop taskstk, dest_str;
call response_update, msg_str, dest_str, $1;
addq msg_str, $dest_str.list, $1;
sizeq $dest_str.wait, br_chk;
if (br_chk > #0);
    delq $dest_str.wait, temp_tcb;
call sched_rte_task, temp_tcb;
else;
    sizeq await, br_chk;
    if (br_chk > #0);
        delq await, temp_tcb;
        if (temp_tcb.msg_in = #1);
            move temp_tcb.msg_in, dest_str;
            sizeq $dest_str.list, br_chk;
            if (br_chk > #0);
                move $2, temp_tcb.msg_in;
call sched_rte_task, temp_tcb;
    endif;
endif;
endif;
*time end_os;
*shb end_os, start_os, start_os;
*add start_os, os_ovhd, os_ovhd;

enable #0;
ENDPAIN;

; PRIMITIVE recv;
;
; This primitive performs a blocking receive operation
PAIN recv;

disable #0;
*time start_os;
pop taskstk, msg_str;
pop taskstk, dest_str;

sizeq $dest_str.list, br_chk;
if (br_chk <> #0);
    delq $dest_str.list, msg_str;
call response_update, msg_str, dest_str, #2;
endif;

declare #;
*time end.os;
*sub end.os, start.os, start.os;
*add start.os, os.ovhd, os.ovhd;
ELSE;
*time curr_tcb sts.end;
*sub curr_tcb sts.end, curr_tcb sts.start, curr_tcb sts.start;
*add curr_tcb sts.start, curr_tcb sts.run_time, curr_tcb sts.run_time;
*move curr_tcb sts.run_time, &run_str;
*time curr_tcb sts.start;
move dest_str, curr_tcb m_wait;
move msg_str, curr_tcb elt_str;
move */wait/, curr_tcb status;
move taskstk, curr_tcb stk;
add #2, pc, curr_tcb pclnt;
addq curr_tcb, &dest_str wait, $1;
call run_str task;
disable #0;
move curr_tcb elt_str, msg_str;
move curr_tcb m_wait, dest_str;
deql &dest_str list, &msg_str;
call response update, msg_str, dest_str, #2;
ENDP;
enable #0;
ENDPRIM;

; PRIMITIVE evr_hnd
;
; This clock event handler returns waiting tasks to the ready to run queue
PRIM evr_hnd;
disable #0;
*sub curr_tcb timer, os os;
*time start os;
*move wait, wu chk;
IF (wu chk > 0); 
*move curr_tcb, temp_tcb;
*move $0, temp_tcb msg in;
*move temp_tcb, temp2_tcb;
*move temp_tcb, curr_tcb;
*call response update, 0, 0;
*move curr_tcb, temp_tcb;
*move temp_tcb, curr_tcb;
call sched txt task, temp_tcb;
ENDP;
*time end os;
*sub end os, start os, start os;
*add start os, os ovhd, os ovhd;
enable #0;
ENDPRIM;

; PRIMITIVE waits
;
; This primitive allows tasks to voluntarily suspend themselves.
PRIM waits;
disable #0;
*time start os;
*pop taskstk, num_parms;
IF (num_parms < 0); 
*pop taskstk, msg_str;
*pop taskstk, dest_str;
ELSE;
*pop taskstk, res pc;
ENDP;
*time curr_tcb sts.end;
*sub curr_tcb sts.end, curr_tcb sts.start, curr_tcb sts.start;
*add curr_tcb sts.start, curr_tcb sts.run_time, curr_tcb sts.run_time;
*move curr_tcb sts.run_time, &run_str;
*time curr_tcb sts.start;
move */wait/, curr_tcb status;
move taskstk, curr_tcb stk;
IF (num_parms <> 0); 
move dest_str, curr_tcb m_wait;
*move msg_str, curr_tcb elt_str;
move $1, curr_tcb msg in;
add #2, pc, curr_tcb pclnt;
addq curr_tcb, wait, curr_tcb priority;
call run_str task;
IF (curr_tcb msg in = 2);
move curr_tcb.alt_str, msg_str;
move curr_tcb.m_wait, dest_str;
delq @dest_str.list, @msg_str;
call response_update, @msg_str, dest_str, #2;
ENDIF;
ELSE;
move $0, curr_tcb.msg_in;
move ret_pc, curr_tcb.pcnt;
addq curr_tcb, sync, curr_tcb.priority;
call run_str_task;
ENDIF;

; PRIMITIVE endresp
;
; This primitive simply prints event traces at the end of a response.
PRIM endresp;
*disable $0;
*call response_update, $0, $5;
*enable $0;
ENDIF;

; PRIMITIVE nearesp
;
; This primitive simply clears a tasks current trace history before it
; starts working on a new response.
PRIM nearesp;
disable $0;
*sizeq curr_tcb.ts.resp_q, q.size;
*WHILE (q.size > $0);
  *delq curr_tcb.ts.resp_q, trace_alt;
  *sizeq curr_tcb.ts.resp_q, q.size;
*ENDWHILE;
*enable $0;
ENDIF;

BEGIN
move #/node/, nod_name;
disable $0;
move tasklist, proclist;
sizeq proclist, br_chk;
move currtask, tempstr;
*move $1, wt_chk;
while br_chk < $0;
  delq proclist, tcb_name;
  crsav tcb_name;
  *time curr_tcb.ts.start;
  move wt_chk, curr_tcb.num;
  *add $1, wt_chk, wt_chk;
  move tasksrt, curr_tcb.pcnt;
  move tcb_name, curr_tcb.name;
  move #/ready/, curr_tcb.status;
  move currprio, curr_tcb.priority;
  move taskstk, curr_tcb.stk;
  addq curr_tcb, rtr, currprio;
  sizeq proclist, br_chk;
endwhile;

crsav tempstr;
move #/wdy_tm/, wdy_str;
move #/we_tm/, we_str;
move #/run_tm/, run_str;
*move #/END RESPONSE ................../, end_str;
bind evt_hnd, #0;
*enable $0;
printer #/Kernel Starting.../;
call run_rtr_task;
printer #/ERROR - KERNEL TERMINATING/;
ENDTASK;
ENDKERNEL;

317
B.1.3 Idle Task Code

\begin{verbatim}
<table>
<thead>
<tr>
<th>Low priority idle task code</th>
</tr>
</thead>
<tbody>
<tr>
<td>STAT rdy_tm, integer;</td>
</tr>
<tr>
<td>STAT ut_tm, integer;</td>
</tr>
<tr>
<td>STAT run_tm, integer;</td>
</tr>
<tr>
<td>BEGIN</td>
</tr>
<tr>
<td>CALLP nilprim;</td>
</tr>
<tr>
<td>SETSTATE #/start/;</td>
</tr>
<tr>
<td>LOOP</td>
</tr>
<tr>
<td>STATESSELECT;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>END;</td>
</tr>
<tr>
<td>ENDSELECT;</td>
</tr>
<tr>
<td>ENDLOOP</td>
</tr>
<tr>
<td>ENDTASK</td>
</tr>
</tbody>
</table>
\end{verbatim}

B.1.4 Initial Application Task Partitioning

The following tasks correspond to the partitioning depicted in figure 6.5 in chapter 6.

\begin{verbatim}
<p>| TASK T1 |</p>
<table>
<thead>
<tr>
<th>Activity A1, A2 Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>STAT rdy_tm, integer;</td>
</tr>
<tr>
<td>STAT ut_tm, integer;</td>
</tr>
<tr>
<td>STAT run_tm, integer;</td>
</tr>
<tr>
<td>VAR  msg1, t_msg, elt;</td>
</tr>
<tr>
<td>VAR  msg2, t_msg, elt;</td>
</tr>
<tr>
<td>VAR  tak_strt, integer;</td>
</tr>
<tr>
<td>VAR  tak_end, integer;</td>
</tr>
<tr>
<td>STAT cycle, integer;</td>
</tr>
<tr>
<td>VAR  b, integer;</td>
</tr>
<tr>
<td>ACTIVITY init;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ENDACT;</td>
</tr>
<tr>
<td>ACTIVITY upd_cycle;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ENDACT;</td>
</tr>
<tr>
<td>ACTIVITY get_rand;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ENDACT;</td>
</tr>
<tr>
<td>ACTIVITY a1;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ENDACT;</td>
</tr>
<tr>
<td>ACTIVITY a2;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ENDACT;</td>
</tr>
<tr>
<td>BEGIN</td>
</tr>
<tr>
<td>DO_ACT init;</td>
</tr>
<tr>
<td>CALLP nilprim;</td>
</tr>
<tr>
<td>SETSTATE #/clear/;</td>
</tr>
<tr>
<td>END;</td>
</tr>
</tbody>
</table>
\end{verbatim}

\begin{verbatim}
<p>| TASK T2 |</p>
<table>
<thead>
<tr>
<th>Activity A3, A4 Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>STAT rdy_tm, integer;</td>
</tr>
<tr>
<td>STAT ut_tm, integer;</td>
</tr>
<tr>
<td>STAT run_tm, integer;</td>
</tr>
<tr>
<td>VAR  msg1, t_msg, elt;</td>
</tr>
<tr>
<td>VAR  msg2, t_msg, elt;</td>
</tr>
<tr>
<td>VAR  msg3, t_msg, elt;</td>
</tr>
<tr>
<td>VAR  msg4, t_msg, elt;</td>
</tr>
<tr>
<td>VAR  cyc_strt, integer;</td>
</tr>
<tr>
<td>VAR  cyc_end, integer;</td>
</tr>
<tr>
<td>STAT cycle, integer;</td>
</tr>
<tr>
<td>ACTIVITY init;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ENDACT;</td>
</tr>
<tr>
<td>ACTIVITY upd_cycle;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ENDACT;</td>
</tr>
<tr>
<td>ACTIVITY a3;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ENDACT;</td>
</tr>
<tr>
<td>ACTIVITY a4;</td>
</tr>
</tbody>
</table>
\end{verbatim}
ticks $200;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
LIST #/start/;
STATESELECT;
STATE clear;
IF $true;
DO_ACT upd_cycle;
CALLP recv, $/q2/,$/msg1/;
SETSTATE #/start/;
ENDIF;
ENDSTATE;
STATE start;
IF $true;
CALLP $/get_2/;
SETSTATE #/wait3/;
ENDIF;
ENDSTATE;
STATE get_2;
IF $true;
DO_ACT send, $/q4/,$/msg2/;
SETSTATE #/wait1/;
ENDIF;
ENDSTATE;
STATE wait;
IF $true;
CALLP recv, $/q1/,$/msg3/;
SETSTATE #/get_1/;
ENDIF;
ENDSTATE;
STATE get_1;
IF $true;
DO_ACT send, $/q3/,$/msg4/;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;
STATESELECT;
ENDLOOP;
ENDTASK;

CALLP nilprim;
SETSTATE #/clear/;
LOOP;
STATESELECT;
STATE clear;
IF $true;
DO_ACT upd_cycle;
CALLP recv, $/q4/,$/msg1/;
SETSTATE #/start/;
ENDIF;
ENDSTATE;
STATE start;
IF $true;
CALLP recv, $/q4/,$/msg2/;
SETSTATE #/got_4/;
ENDIF;
ENDSTATE;
STATE got_4;
IF $true;
DO_ACT send, $/q4/,$/msg3/;
SETSTATE #/wait1/;
ENDIF;
ENDSTATE;
STATE wait3;
IF $true;
CALLP recv, $/q3/,$/msg2/;
SETSTATE #/got_3/;
ENDIF;
ENDSTATE;
STATE got_3;
IF $true;
DO_ACT send, $/q4/,$/msg4/;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;
STATESELECT;
ENDLOOP;
ENDTASK;

B.1.5 Second Application Task Partitioning

The following tasks correspond to the partitioning depicted in figure 6.6 in chapter 6.

TASK T1
Activity A1, A2, A5 Code

CALLP nilprim;
SETSTATE #/clear/;
LOOP;
STATESELECT;
STATE clear;
IF $true;
DO_ACT upd_cycle;
CALLP recv, $/q4/,$/msg1/;
SETSTATE #/start/;
ENDIF;
ENDSTATE;
STATE start;
IF $true;
CALLP recv, $/q4/,$/msg2/;
SETSTATE #/got_4/;
ENDIF;
ENDSTATE;
STATE got_4;
IF $true;
DO_ACT send, $/q4/,$/msg3/;
SETSTATE #/wait1/;
ENDIF;
ENDSTATE;
STATE wait3;
IF $true;
CALLP recv, $/q3/,$/msg2/;
SETSTATE #/got_3/;
ENDIF;
ENDSTATE;
STATE got_3;
IF $true;
DO_ACT send, $/q4/,$/msg4/;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;
STATESELECT;
ENDLOOP;
ENDTASK;

319
ACTIVITY a1;
ticks $200;
ENDACT;

ACTIVITY a2;
ticks $200;
ENDACT;

ACTIVITY a5;
ticks $200;
moves $0, got_msg;
ENDACT;

BEGIN
DO ACT init;
CALLP nilprim;
SETSTATE #/clear/
ENDLOOP
ENDTASK;

| TASK T2 |
| Activity A3, A6 Code |

STATL rdy_tm, integer;
STAT wu_tm, integer;
STAT run_tm, integer;
VAR msg1, t_msg1_alt;
VAR msg2, t_msg2_alt;
VAR msg3, t_msg3_alt;
VAR msg4, t_msg4_alt;
VAR cyc_strt, integer;
VAR cyc_end, integer;
STAT cycle, integer;

BEGIN
DO ACT init;
CALLP nilprim;
SETSTATE #/clear/
ENDLOOP

ENDACT;

ACTIVITY init;

*time cyc_strt;
ENDACT;

ACTIVITY upd_cycle;

*time cyc_end;
*sub cyc_end, cyc_strt, cycle;
*time cyc_start;
ENDACT;

ACTIVITY a3;
ticks $200;
ENDACT;

ACTIVITY a4;
ticks $200;
ENDACT;

BEGIN
DO ACT init;
CALLP nilprim;
SETSTATE #/clear/
ENDLOOP

ENDACT;

STATE select;
IF #true;
DO ACT
CALLP
SETSTATE
# / start /
ENDIF;
ENDSTATE;

STATE start1;
IF #true;
CALLP
wait $ / get_msg $2 ;
SETSTATE
# / event /
ENDIF;
ENDSTATE;

STATE event1;
IF (got_msg $2);
DO ACT
get
CALLP
SETSTATE
# / start /
ENDIF;
ENDSTATE;

STATE act1;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act2;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act3;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act4;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act5;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act6;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act7;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act8;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act9;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act10;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act11;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act12;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act13;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act14;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act15;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act16;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act17;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act18;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act19;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act20;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act21;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act22;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act23;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act24;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act25;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act26;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act27;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act28;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act29;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;

STATE act30;
IF #true;
DO ACT
CALLP
SETSTATE
# / get msg /
ENDIF;
ENDSTATE;
B.1.6 Third Application Task Partitioning

The following tasks correspond to the partitioning depicted in figure 6.7 in chapter 6.

```plaintext
TASK T1
Activity A1, A2 Code

STAT rdy_tm, integer;
STAT wt_tm, integer;
STAT run_tm, integer;

VAR msg1, t_msg, elt;
VAR msg2, t_msg, elt;
VAR tak_strt, integer;
VAR tak_end, integer;
STAT cycle, integer;
VAR b, integer;

ACTIVITY init;
 TIME tak_strt;
ENDACT;

ACTIVITY upd_cycle;
 TIME tak_end;
 +sub tak_end, tak_strt, cycle;
 +time tak_strt;
ENDACT;

ACTIVITY get_rand;
 rand #1, b;
ENDACT;

ACTIVITY a1;
 ticks #200;
ENDACT;

ACTIVITY a2;
 ticks #200;
ENDACT;

BEGIN
DO ACT init;
CALLP nilprim;
SETSTATE #/clear/;
ENDDO;
LOOP;
STATESELECT;

STATE clear;
 IF #true;
 DO ACT
 CALLP
upd_cycle;
newresp;
SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
 IF #true;
 CALLP
 waits, #0;
SETSTATE #/event/;
ENDIF;
ENDSTATE;

STATE event;
 IF #true;
 DO ACT
 CALLP
 get_rand;
send, #/q1/, #/msg1/;
SETSTATE #/acts/;
ENDIF;
ENDSTATE;

STATE resp;
 IF #true;
 CALLP
 endresp;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

STATE acts;
 IF (#0);
 CALLP
 nilprim;
SETSTATE #/act1/;
ENDIF;
IF (#1);
 CALLP
 nilprim;
SETSTATE #/act2/;
ENDIF;
ENDSTATE;

STATE act1;
 IF #true;
 DO ACT
 a1;
 CALLP
 send, #/q1/, #/msg2/;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

STATE act2;
 IF #true;
 DO ACT
 a2;
 CALLP
 send, #/q1/, #/msg2/;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

ENDSELECT;
ENDLOOP
ENDTASK;

TASK T2
Activity A3, A4, A5 Code

STAT rdy_tm, integer;
STAT wt_tm, integer;
STAT run_tm, integer;

VAR msg1, t_msg, elt;
VAR msg2, t_msg, elt;
VAR msg3, t_msg, elt;
VAR msg4, t_msg, elt;
VAR cyc_strt, integer;
VAR cyc_end, integer;
STAT cycle, integer;

ACTIVITY init;
 +time cyc_strt;
ENDACT;

ACTIVITY upd_cycle;
 +time cyc_end;
 +sub cyc_end, cyc_strt, cycle;
 +time cyc_strt;
ENDACT;

ACTIVITY a3;
 ticks #10;
ENDACT;

ACTIVITY a4;
 ticks #10;
ENDACT;

ACTIVITY a5;
 ticks #10;
ENDACT;

BEGIN
DO ACT init;
CALLP nilprim;
SETSTATE #/clear/;
ENDDO;
LOOP;
STATESELECT;

STATE clear;
 IF #true;
 DO ACT
 CALLP
 upd_cycle;
ENDIF;
ENDSTATE;

STATE end;
 IF #true;
 CALLP
endresp;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

STATE acts;
 IF (#0);
 CALLP
 nilprim;
SETSTATE #/act1/;
ENDIF;
IF (#1);
 CALLP
 nilprim;
SETSTATE #/act2/;
ENDIF;
ENDSTATE;

STATE act1;
 IF #true;
 DO ACT
 a1;
 CALLP
 send, #/q1/, #/msg2/;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

STATE act2;
 IF #true;
 DO ACT
 a2;
 CALLP
 send, #/q1/, #/msg2/;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

ENDSELECT;
ENDLOOP
ENDTASK;
```

321
CALLP
SETSTATE
ENDIF;
ENDSTATE;

STATE start;
IF #true;
CALLP
SETSTATE
ENDIF;
ENDSTATE;

STATE wait;
IF #true;
DO_ACT
CALLP
SETSTATE
ENDIF;
ENDSTATE;

STATE wait2;
IF #true;
DO_ACT
CALLP
SETSTATE
ENDIF;
ENDSTATE;

STATE dim;
IF #true;
DO_ACT
CALLP
SETSTATE
ENDIF;
ENDSTATE;

ENDSELECT;
ENDLOOP;
ENDTASK;

neursp;
#/start/;

recv, #/q2/, #/msg1/;
#/wait/;

nd;
recv, #/q1/, #/msg3/;
#/wait2/;

a3;
nalprin;
#/dim/;

a6;
endresp;
#/clean/;
B.2 Hardware Platform Analysis

This section contains code for the hardware platform determination process. The kernel, and application task files were the same for all prototypes. The only changes involved moving tasks within the node files to generate different task to processor allocations.

B.2.1 Example Node File

```
<table>
<thead>
<tr>
<th>NODE FILE</th>
</tr>
</thead>
</table>

| t_trace_elt.act Values: | \#1 - Msg Sent |
|                       | \#2 - Msg Accepted |
|                       | \#3 - Shared Data Write |
|                       | \#4 - Shared Data Read |
|                       | \#5 - Response Out |
|                       | \#6 - Event Trigger |
|                       | \#7 - Start Read Delimiter |
|                       | \#8 - End Read Delimiter |
|                       | \#9 - Start Mag Delimiter |
|                       | \#10 - End Mag Delimiter |

TYPE t_trace_elt = node_name:string *
    task_name:string *
    act:integer *;
    Action
    time:integer *;
    Time of action
    misc1:integer *;
    misc2:integer *;

TYPE t_mag_elt = data:integer *;
    type:integer *;
    src:string *;
    trc_size:integer;
    No. of associated trace els

VAR node_name, string;
    symbolic node name

VAR mag.ryd, integer;
    used for external messages by MSG

VAR magtype, integer;
    used for external messages by MSG

VAR synch.loss, integer;
    used by SYNCH

VAR sig.ryd, integer;
    used for signalling messages by SIGNAL

VAR sigtype, integer;
    used for signalling messages by SIGNAL

EVENT clk1;

KERNEL base1;

TASK mag, 10;
    TASK signal, 10;
    TASK audits, 10;
    TASK diag, 10;
    TASK tondere, 10;
    TASK chnlding, 10;
    TASK msgdiag, 10;
    TASK tidng, 10;
    TASK timto, 10;
    TASK idle, 1;

ENDNODE;
```
B.2.2 Kernel File

KERNEL CODE FILE

This kernel contains the following primitives:
- nonblocking send
- blocking selective receive
- sleep
- get data from another task
- nilprime

The following external message types are supported:
1 - start call setup
2 - provide tone
3 - run diagnostic
4 - integrity ok
5 - integrity failure
6 - call takedown

The following internal message types are supported:
1 - START - request call setup
2 - TONE5 - request for tone
3 - DIAG - request single random diagnostic
4 -
5 -
6 -
7 - MSG - report fault
8 - MSG - perform and report synch operation
9 - FAULT - synch loss
10 - FAULT - general fault report
11 - CONNECT - perform connect/disconnect
12 - CEHLS - allocate channel
13 - CEHLS - free channel
14 - ALL TASKS - channel granted
15 - STAT - call setup complete
16 - SIGNAL - start integrity checking
17 - STAT - Integrity Failure report
18 - END - caller hung up
19 - STAT - call finished
20 - STAT - stop Integrity checking
21 - ALL DIALS - run diagnostic
22 - TIMTC - step carrier checking
23 - TIMTC - monitor carriers

The following response types are supported:
1 - MSG - fault report sent to switching center
2 - MSG - synch source changed
3 - CONNECT - switching center tone request connected
4 - STAT - call finished
5 - STAT - call started
6 - ALL DIALS - diagnostic completed without failure
7 - TIMTC - carrier scan completed without failure

KERNEL base;

TASK kern_task, 50;

TYPE t_tcb_state = rdy_time : integer *;
run_time : integer *;
wt_time : integer *;
begin : integer *;
finish : integer *;
resp_q : queue ; of t_trace_elt

TYPE t_shared_state = data_trace :queue ; of t_trace_elt

TYPE t_tcb = name : string *;
num : integer *;
status : string * ; task state
priority : integer * ; priority
pent : integer * ; program counter
ele_str : string * ; rcv msg elt
stk : stack * ; local task stack
sts : t_tcb_state ; task control block

TYPE t_sh_data = task_num : string *
num_msgs : integer * ; number of queued messages
msgs : queue * ; message queue (t_msg_elt)
trcs : queue ; trace queue (t_trace_elt)

TYPE t_msg_struct = task_num : string *
waiting : integer * ; flag set if blocked for msg

324
msg_q : queue *
traces : queue *
type : integer *
block : t_tcb

TYPE t_id_struct = nom : string *
trc_name : string *
st_name : string *

SHARED comm_q, queue; of t_id_struct (one elt per task)
SHARED sh_h1, t_sh_data;
SHARED sh_h2, t_sh_data;
SHARED sh_h3, t_sh_data;
SHARED sh_h4, t_sh_data;
SHARED sh_h5, t_sh_data;
SHARED sh_h6, t_sh_data;
SHARED sh_h7, t_sh_data;
SHARED sh_h8, t_sh_data;
SHARED sh_h9, t_sh_data;
SHARED sh_h10, t_sh_data;
SHARED sh_h11, t_sh_data;
SHARED sh_h12, t_sh_data;
SHARED sh_h13, t_sh_data;
SHARED sh_h14, t_sh_data;
SHARED sh_h15, t_sh_data;

; Message queues
VAR loc_comm_q, queue; of t_id_struct (one elt per local task)
VAR st_elt, t_id_struct;
VAR L1_elt, t_mag_struct;
VAR L2_elt, t_mag_struct;
VAR L3_elt, t_mag_struct;
VAR L4_elt, t_mag_struct;
VAR L5_elt, t_mag_struct;
VAR L6_elt, t_mag_struct;
VAR L7_elt, t_mag_struct;
VAR L8_elt, t_mag_struct;
VAR L9_elt, t_mag_struct;
VAR L10_elt, t_mag_struct;
VAR L11_elt, t_mag_struct;
VAR L12_elt, t_mag_struct;
VAR L13_elt, t_mag_struct;
VAR L14_elt, t_mag_struct;
VAR L15_elt, t_mag_struct;
VAR L16_elt, t_mag_struct;
VAR L17_elt, t_mag_struct;
VAR local, string;
VAR shared, string;
VAR loc_trc, string;
VAR sh_trc, string;

STAT sends, integer;
STAT os_evh, integer;
STAT switches, integer;
STAT evt_0, integer;
STAT evt_3, integer;
STAT evt_4, integer;
VAR start_os, integer;
VAR end_os, integer;
VAR elapsed, integer;
VAR rd_s, string;
VAR ut_s, string;
VAR run_s, string;
VAR end_s, string;

VAR swait, queue;
VAR ut_chk, integer;
VAR msg_s, string;
VAR dest_s, string;
VAR temp_tcb, t_tcb;
VAR temp_tcb, t_tcb;
VAR ret_pc, integer;
VAR tmp_stk, stack;
VAR num_parms, integer;

325
VAR found, integer;
VAR msg_type, integer;
VAR data_size, integer;
VAR nil_queue, queue;
VAR nil_string, string;
VAR msg_elt, t_msg_elt;
VAR m_num, integer;
VAR temp_q, queue;
VAR type, integer;
VAR q_size, integer;
VAR done, integer;

; Global Kernel Variables
VAR cur_tcb, t_tcb; ; current task control block
VAR proclist, queue; ; copy of task name list
VAR br_chk, integer; ; branch condition check
VAR tmpstr, string; ; temporary string value
VAR tcb_name, string; ; current task name;
VAR rtr, queue; ; ready to run queue;

; Variables for get_local and get_sh procedures
VAR g_q_size, integer;
VAR g_found, integer;
VAR g_at_elts, t_id_struct;

; Variables for response_update procedure
VAR action, integer; ; action requested in response_update
VAR prio, integer; ; send priority
VAR trace, t_trace_elt; ; new response trace element
VAR msg_elt_name, string; ; name of msg elt for response trace
VAR dest_elt_name, string; ; name of msg queue
VAR delim_elt, t_trace_elt; ; msg or data trace delimiter
VAR q_size, integer; ; size of trace queue
VAR data_name, string; ; name of shared data variable
VAR temp_q, queue; ; data response queue copy

; VARIABLE declarations for procedure NUM_RTR_TASK
VAR error, boolean; ; Set if no ready processes

; VARIABLE declarations for procedure SCHED_RTR_TASK
VAR rtr_pcb, t_tcb; ; PCB to be scheduled

; PROCEDURE response_update (Gname : string;
; action: integer)
;
; This procedure updates response trace queues
PROC response_update;
*pop taskstk, action;
*move mod_name, trace_elt.node_name;
*move cur_tcb_name, trace_elt.task_name;
*addq trace_elt.time, trace_elt.time;
*move action, trace_elt.act;
*if (action = $1); SEND MSG
*pop taskstk, prio;
*pop taskstk, dest elt name;
*addq trace elt, cur_tcb.sts.resp_q, $1;
*move cur_tcb.sts.resp_q, temp q;
*addq temp q, q size;
*move q size, msg elt.name.trc size;
*while (q size > %0);
;delq temp q, trace elt;
*addq trace elt, delim elt name, prio;
*addq temp q, q size;
*endwhile;
*endif;
*if (action = #2); RCV MSG
*pop taskstk, dest elt name;
*pop taskstk, msg elt name;
*move trace elt, delim elt;
*addq trace elt, cur_tcb.sts.resp_q, $1;
*move $0, delim elt.act;

326
PROCEDURE run_rtr_task

; This procedure takes the first ready to run task off the rtr queue (this is the one with the highest priority) and copies its saved state into the current processor state. If there are no ready to run tasks, then an error message is printed and the kernel terminates.

PROC run_rtr_task;
  sizeq rtr, br_chk;
  if (br_chk = 0);
  printer #ERROR: NO RTR PROCESSES/;
  halt;
ENDPROC;

*addq delim_elt, curr_tcb.ste.resp_q, #1;
*move msg_delim_elt, curr_tcb.ste.resp_q, #1;
*while (q_size > 0);
  *delq dest_delim_elt, trace_elt;
  *addq trace_elt, curr_tcb.ste.resp_q, #1;
  *sub q_size, #1, q_size;
  *endwhile;

*move #10, delim_elt.isact;
*addq delim_elt, curr_tcb.ste.resp_q, #1;
*endif;

*if (action = #3) DATA WRITE
  *pop taskstk, data_name;
  *addq trace_elt, curr_tcb.ste.resp_q, #1;
  *move curr_tcb.ste.resp_q, data_name.stats.trace;
*endif;

*if (action = #4) DATA READ
  *move trace_elt, delim_elt;
  *addq trace_elt, curr_tcb.ste.resp_q, #1;
  *move #7, delim_elt.isact;
  *addq delim_elt, curr_tcb.ste.resp_q, #1;
  *move data_name.stats.data_trace, temp_q;
  *sizeq temp_q, q_size;
  *while (q_size > 0);
    *delq temp_q, trace_elt;
    *addq trace_elt, curr_tcb.ste.resp_q, #1;
    *sizeq temp_q, q_size;
  *endwhile;

*move #6, delim_elt.isact;
*addq delim_elt, curr_tcb.ste.resp_q, #1;
*endif;

*if (action = #6) RESPONSE OUT
  *pop taskstk, trace_elt.misc1;
  *addq trace_elt, curr_tcb.ste.resp_q, #1;
  *sizeq curr_tcb.ste.resp_q, q_size;
  *while (q_size > 0);
    *delq curr_tcb.ste.resp_q, trace_elt;
    *stypc #/task: /;
    *stypc trace_elt.task_name;
    *stypc #/ action: /;
    *stypc trace_elt.isact;
    *stypc #/ time: /;
    *stypc trace_elt.time;
    *stypc #/ misc1: /;
    *stypc trace_elt.misc1;
    *stypc #/ misc2: /;
    *stypc trace_elt.misc2;
  *sizeq curr_tcb.ste.resp_q, q_size;
  *endwhile;
*stypcor end_str;
*endif;

*if (action = #8) EVENT TRIGGER
  *pop taskstk, trace_elt.misc1;
  *setime trace_elt.misc2, trace_elt.misc1;
  *addq trace_elt, curr_tcb.ste.resp_q, #1;
*endif;
ENDPROC;
ELSE;
  mov rtr, curr_tcb;
  mov cur_tcb_name, tcb_name;
  add $1, switches, switches;
  mov curr_tcb_sts.finish;
  sub curr_tcb_sts.finish, curr_tcb_sts.begin, curr_tcb_sts_begin;
  mov $/rdy_str, rdy_str;
  mov curr_tcb_sts_rdy, &rdy_str;
  mov curr_tcb_sts_run_time, &str_str;
  mov curr_tcb_sts_begin;
  mov $/running, curr_tcb_sts.status;
  mov curr_tcb_stk, taskstk;
  mov end_os;
  sub end_os, start_os, elapsed;
  add elapsed, os_ovhd, os_ovhd;
  mov $10;
  mov curr_tcb_pcnt, pc;
ENDIF;
ENDPAC;

; PROCEDURE sched_rtr_task (rtr_pcb : t_pcb)
; This procedure accepts a process control block and adds it to
; the ready to run queue in a prioritized fashion.
PROC sched_rtr_task;
pop taskstk, rtr_pcb;
  mov $/ready, rtr_pcb_status;
  mov curr_tcb_sts.finish;
  sub curr_tcb_sts.finish, curr_tcb_sts_begin, curr_tcb_sts_begin;
  add curr_tcb_sts_begin, curr_tcb_sts_run_time, curr_tcb_sts_run_time;
  mov curr_tcb_sts_run_time, &run_str;
  mov curr_tcb_sts_begin;
  mov $/ready, curr_tcb_sts.status;
  addq curr_tcb, rtr, curr_tcb_priority;
  call run_rtr_task;
  halt;
endif;
ENDPAC;

; PROCEDURE get_sh (taskname : string)
; This procedure accepts a task name and sets the shared and
; sh_tcb variables to hold the names of the task's messaging and trace
; structures.
PROC get_sh;
pop taskstk, tempstr;
  mov $0, g_found;
  lock;
senq comm_q, g_q_size;
  while (g_q_size > 0) & (~g_found);
  delq comm_q, g_st_elts;
    if (g_st_elts.name = tempstr);
      mov $1, g_found;
      endif;
    addq g_st_elts, comm_q, $1;
  sub g_q_size, $1, g_q_size;
  endwhile;
unlock;
  if (g_found = $1);
    mov g_st_elts_name, shared;
    mov g_st_elts_tnr_name, sh_tcb;

328
else;
    move nil_string, shared;
    move nil_string, sh_trc;
endif;
ENDPROC;

PROCEDURE get_local (taskname : string)
This procedure accepts a task name and sets the local and
loc_trc variables to hold the names of the task's messaging and trace
structures.

PARG get_local;
pop taskstk, tempstr;
move $0, g_found;
sneq loc_comm_q, g_q.size;
while (g_q.size > $0) & ('g_found');
delq loc_comm_q, g_st_elt;
    if (g_st_elt.nmc = tempstr);
        move $1, g_found;
    endif;
    addq g_st_elt, loc_comm_q, $1;
    sub g_q.size, $1, g_q.size;
endwhile;

if (g_found = $1);
    move g_st_elt.st_name, local;
    move g_st_elt.trc_name, loc_trc;
else;
    move nil_string, local;
    move nil_string, loc_trc;
endif;
ENDPROC;

PRIMITIVE nilprim
This dummy primitive does nothing except pull in internode
messages.

PARG nilprim;
disable $0;
*time start_cw;
move curr_tcb.stats.rdy_time, rdy_str;
*move curr_tcb.stats vt_time, Get_str;
*move curr_tcb.stats run_time, run_str;
if (num_queued > $0);
    move $0, done;
sneq loc_comm_q, q_ms;
while (q_ms > $0) & ('done');
delq loc_comm_q, st_elt;
    move st_elt.st_name, local;
call get_sh, st_elt.nmc;
lock;
while (gshared.num_msgs > $0);
    sub num_queued, $1, num_queued;
    sub gshared.num_msgs, $1, gshared.num_msgs;
delq gsharedmsgs, msg elt;
    if (glocal.waiting = $1);
        if (glocal.vt.type = msg elt.type);
            addq msg elt, glocal.msg q, $2;
            while (msg elt.trc size > $0);
                *delq gsharedmsgs, msg elt;
                *addq msg elt, glocal.trcs, $2;
                add msg elt.trc size, $1, msg elt.trc size;
        endwhile;
        else;
            addq msg elt, glocal.msg q, $1;
            while (msg elt.trc size > $0);
                *delq gsharedmsgs, msg elt;
                *addq msg elt, glocal.trcs, $1;
                add msg elt.trc size, $1, msg elt.trc size;
        endwhile;
        endif;
    if (glocal.vt.type = $0) | (glocal.vt.type = msg elt.type);
        move $0, glocal.waiting;
        move $1, done;
    endif;
endif;
endif;

329
endwhile;
unlock;
addr st.alt, loc.comm, #1;
sub q.sz, #1, q.sz;
endwhile;
if (done = #1);
call sched_xr., task, @local.block;
endif;
edif;
enable #10;
ENDPRIM;

! PRIMITIVE send (task_name : string; @msg : string)
!
! This primitive accepts the name of a destination task and a
! message pointer. It then performs a nonblocking send operation.
PRIM send:
disable #10;
*add #1, sends, sends;
*time start,s;
pop taskstr, msg_str;
pop taskstr, dest_str;
call get_local, dest_str;
if (local <> nil_string);
if (@local.vt.type = @msg.type);
call response_update, msg_str, loc_txc, #2, #1;
addr @msg_str, @local.msg.q, #2;
else;
call response_update, msg_str, loc_txc, #1, #1;
addr @msg_str, @local.msg.q, #1;
endif;
if (@local.vt.type = #0) | (@local.vt.type = @msg.type));
move #0, @local.waiting;
call sched_xr., task, @local.block;
endif;
else;
call response_update, msg_str, loc_txc, #1, #1;
addr @msg_str, @local.msg.q, #1;
endif;
else;
call get_sh, dest_str;
lock;
addr #1, num_queued, num_queued;
call response_update, msg_str, sh_txc, #1, #1;
addr @msg_str, @shared.msgs, #1;
addr #1, @shared.nummsgs, @shared.nummsgs;
unlock;
endif;
*time end.s;
*sub end.s, start.s, elapsed.s;
*add elapsed.s, os_ovhd, os_ovhd;
enable #10;
ENDPRIM;

! PRIMITIVE recv (msg.type : integer; @msg : string)
!
! This primitive performs a blocking selective receive operation.
! It accepts the type of message expected (#0 means accept any type)
! and a pointer to where the message is to be placed. If a
! specific receive type is specified, the task is blocked until
! that type is available. Otherwise, the calling task is blocked
! until any message becomes available.
PROC block_recv:
*time curr_tcb.sts.finish;
*sub curr_tcb.sts.finish, curr_tcb.sts.begin, curr_tcb.sts.begin;
*add curr_tcb.sts.begin, curr_tcb.sts.run_time, curr_tcb.sts.run_time;
move curr_tcb.sts.run_time, @run_str;
*time curr_tcb.sts.begin;
move @msg.type, @local.vt.type;
move #1, @local.waiting;
move msg_str, curr_tcb.alt.str;
move #/wait/, curr_tcb.status;

330
move taskstk, curr_tcb.stk;
add $2, pc, curr_tcb.pcon;
mov curr_tcb, Global.block;
call run_ttr_task;
disable #10;
mov curr_tcb.alt_str, msg_str;
call get_local, currtask;
move $0, @local.waiting;
sizeq @local.mag_q, done;
if (done > $0);
delq @local.mag_q, msg_str;
call response_update, msg_str, loc_ttc, $2;
endif;

PRIM recv;
disable #10;
*time start-os;
pop taskstk, msg_str;
pop taskstk, msg_type;
call get_local, currtask;
call get_sh, currtask;
while (@shared.num_msgs > $0);
lock;
sub num_queued, $1, num_queued;
sub @shared.num_msgs, $1, @shared.num_msgs;
delq @sharedmsgs, msg_elt;
addq msg_elt, @local.mag_q, $1;
while (msg_elt.trc_size > $0);
delq @share痕迹, trace_elt;
addq trace_elt, @local.traces, $1;
sub msg_elt.trc_size, $1, msg_elt.trc_size;
endwhile;
unlock;
endwhile;
move $0, found;
sizeq @local.mag_q, q.size;
if (q.size ≠ $0);
call block_rcv;
else;
move $1, found;
if (msg_type ≠ $0);
; reorder msg queue because receive specific
move nil_queue, temp_q;
move nil_queue, temp2_q;
move $0, found;
sizeq @local.mag_q, m_num;
while (m_num > $0);
delq @local.mag_q, msg_elt;
if (*found & (msg.alt.type = msg_type);
move $1, found;
addq msg_elt, temp_q, $2;
while (msg_elt.trc_size > $0);
delq @local.traces, trace_elt;
addq trace_elt, temp2_q, $2;
sub msg_elt.trc_size, $1, msg_elt.trc_size;
endwhile;
else;
addq msg_elt, temp_q, $1;
while (msg_elt.trc_size > $0);
delq @local.traces, trace_elt;
addq trace_elt, temp2_q, $1;
sub msg_elt.trc_size, $1, msg_elt.trc_size;
endwhile;
endif;
sizeq @local.mag_q, m_num;
endwhile;
move temp_q, @local.mag_q;
move temp2_q, @local.traces;
endif;
if (*found);
call block_rcv;
else;
delq @local.mag_q, msg_str;
call response_update, msg_str, loc_ttc, $2;
*time end-os;
sub end_os, start_os, elapsed;

331
PRIMITIVE sleep

This primitive simply blocks the calling task by putting it in
the sleep wait queue.

PRIM sleep:
disable $10;
*time start_os;
pop taskstk, ret_pc;
*time curr_tcb.stc.finish;
*sub curr_tcb.stc.finish, curr_tcb.stc.begin, curr_tcb.stc.begin;
*add curr_tcb.stc.begin, curr_tcb.stc.run_time, curr_tcb.stc.run_time;
*move curr_tcb.stc.run_time, @run_stc;
*time curr_tcb.stc.begin;
move $/wait/, curr_tcb.status;
move taskstk, curr_tcb.stk;
move ret_pc, curr_tcb.pcnt;
addq curr_tcb, wait, curr_tcb.priority;
call run_trx_task;
ENDPRI;

PRIMITIVE get_data (task_name : string; data_name : string; size : integer)

This primitive simulates data access between tasks. If the
specified source task is not resident on the processor, then a number
of shared accesses are made. One access is made for every 100 bytes
of data requested.

PRIM get_data:
disable $10;
*time start_os;
pop taskstk, data_size;
pop taskstk, msg_str;
pop taskstk, dest_str;
call get_local, dest_str;
if (local = nil_string);
while (data_size > 0);
lock;
move msg_str, q.size;
unlock;
sub data_size, $100, data_size;
endwhile;
else;
while (data_size > 0);
move q.size, q.size;
sub data_size, $100, data_size;
endwhile;
endif;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_ovhd, os_ovhd;
enable $10;
ENDPRI;

PRIMITIVE end_resp

This primitive simply calls response update to print the
response traces.

PRIM endresp:
disable $10;
*pop taskstk, m_num;
call response_update, m_num, $5;
enable $10;
ENDPRI;

PRIMITIVE new_resp

This primitive clears the calling task's current trace queue.

PRIM newresp:
disable $10;
; PRIMITIVE evt_sync
; This primitive serves as the interrupt handler for the synchronization
; primitive. This interrupt is raised periodically every 5 seconds.
PRIM evt_sync;
  disable $10;
  ackint $0;
  add $1, evt_0, evt_0;
  time start_os;
  sizeq wait, wt_chk;
  move $0, found;
  while ((wt_chk > $0) & (~found));
    delq wait, temp_tcb;
    if (temp_tcb.name = $/sych/);
      move $1, found;
    else;
      addq temp_tcb, wait, temp_tcb.priority;
      sub wt_chk, $1, wt_chk;
    endif;
  endwhile;
  if (found);
    move curr_tcb, temp2_tcb;
    move temp_tcb, curr_tcb;
    call response_update, $2, $6;
    move curr_tcb, temp_tcb;
    move temp_tcb, curr_tcb;
    call sched_rtr_task, temp_tcb;
  endif;
  time end_os;
  sub end_os, start_os, elapsed;
  add elapsed, os_ovhd, os_ovhd;
  enable $10;
ENDPRIM;

; PRIMITIVE evt_loss_sync
; This primitive serves as the interrupt handler for the synchronization
; loss primitive. This interrupt is raised sporadically to signal a loss of
; synchronization.
PRIM evt_loss_sync;
  disable $10;
  ackint $1;
  add $1, evt_1, evt_1;
  time start_os;
  move $1, synch_loss;
  sizeq wait, wt_chk;
  move $0, found;
  while ((wt_chk > $0) & (~found));
    delq wait, temp_tcb;
    if (temp_tcb.name = $/sych/);
      move $1, found;
    else;
      addq temp_tcb, wait, temp_tcb.priority;
      sub wt_chk, $1, wt_chk;
    endif;
  endwhile;
  if (found);
    move curr_tcb, temp2_tcb;
    move temp_tcb, curr_tcb;
    call response_update, $1, $6;
    move curr_tcb, temp_tcb;
    move temp_tcb, curr_tcb;
    call sched_rtr_task, temp_tcb;
  endif;
  time end_os;
  sub end_os, start_os, elapsed;
  add elapsed, os_ovhd, os_ovhd;
  enable $10;
ENDPRIM;
; PRIMITIVE evt_msg_rx
;
; This primitive serves as the interrupt handler for the incoming external
; message interrupt. This interrupt is raised sporadically to signal the
; arrival of a message. It makes the MSG task ready to run from the
; blocked receive state.

PRIV EVT_MSG_RX:
; disable 10;
; ackint #2;
; *add #1, evt_2, evt_2;
*time start_cs;
move #1, msg_rdy;
rand #10, wt_chk;
if (wt_chk < #5);
move #1, msgtype;
else;
if (wt_chk < #3);
move #2, msgtype;
else;
move #3, msgtype;
endif;
endif;
move #0, done;
call get_local, #/msg/;
if (#local.waiting = #1);
move #0, local.waiting;
*move curr_tcb, temp_tcb;
*move local.block, curr_tcb;
*call response_update, #3, #6;
*move curr_tcb, local.block;
*move temp_tcb, curr_tcb;
move #1, done;
endif;
if (done = #1);
call sched_ext_task, local.block;
endif;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_cvhd, os_cvhd;
enable #10;
ENDPRIV;

; PRIMITIVE EVT_SIGNAL
;
; This primitive serves as the interrupt handler for the signalling
; interrupt handler. This interrupt is raised sporadically to signal the
; arrival of a signalling event. It makes the SIGNAL task ready to run
; from the blocked receive state.

PRIV EVT_SIGNAL:
; disable 10;
; ackint #3;
; *add #1, evt_3, evt_3;
*time start_cs;
move #1, sig_rdy;
rand #10, wt_chk;
if (wt_chk < #3);
move #6, sigtype;
else;
if (wt_chk < #3);
move #4, sigtype;
else;
move #5, sigtype;
endif;
endif;
move #0, done;
call get_local, #/signal/;
if (#local.waiting = #1);
move #0, local.waiting;
*move curr_tcb, temp_tcb;
*move local.block, curr_tcb;
*call response_update, #3, #6;
*move curr_tcb, local.block;
*move temp_tcb, curr_tcb;
move #1, done;
endif;
if (done = #1);
call sched_rtr_task, 0, local_block;
endif;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_ovhd, os_ovhd;
enable $10;
ENDPRIM;

; PRIMITIVE evt_clk
;
; This primitive serves as the interrupt handler for the clock interrupt
; used as a timing base for the audits task. This is a periodic interrupt.
PRIM evt_clk;
disable $10;
ackint #4;
*add $1, evt_4, evt_4;
*time start_es;
*iseq wait, wt_chk;
move $0, found;
while ((wt_chk > $0) & (~found));
deq wait, temp_tcb;
if (temp_tcb.name = #/audits/);
move $1, found;
else;
addq temp_tcb, wait, temp_tcb.priority;
sub wt_chk, #1, wt_chk;
endif;
endwhile;
if (found);
*move curr_tcb, temp_tcb;
*move temp_tcb, curr_tcb;
*call response_update, $4, $6;
*move curr_tcb, temp_tcb;
*move temp_tcb, curr_tcb;
call sched_rtr_task, temp_tcb;
endif;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_ovhd, os_ovhd;
enable $10;
ENDPRIM;

BEGIN
move #/nodes/, nod_name;
disable $10;
move #/l_11/, st_elt.st_name;
move #/l_21/, st_elt.st_name;
move #/l_31/ traces/, st_elt.trc_name;
addq st_elt, loc_comm_q, $1;
move #/l_12/, st_elt.st_name;
move #/l_22/, st_elt.st_name;
addq st_elt, loc_comm_q, $1;
move #/l_32/ st_elt.st_name;
move #/l_13/ st_elt.st_name;
move #/l_23/ st_elt.st_name;
move #/l_33/ st_elt.st_name;
move #/l_14/ st_elt.st_name;
addq st_elt, loc_comm_q, $1;
move #/l_15/ st_elt.st_name;
move #/l_25/ st_elt.st_name;
addq st_elt, loc_comm_q, $1;
move #/l_16/ st_elt.st_name;
move #/l_26/ st_elt.st_name;
addq st_elt, loc_comm_q, $1;
move #/l_17/ st_elt.st_name;
move #/l_27/ st_elt.st_name;
addq st_elt, loc_comm_q, $1;
move #/l_18/ st_elt.st_name;
move #/l_28/ st_elt.st_name;
addq st elt, loc_comm_q, $1;
move #/l_19/ st_elt.st_name;
move #/l_29/ st elt.st_name;
addq st elt, loc_comm_q, $1;
move #/l_30/ st elt.st_name;
move #/l 30/ traces/, st_elt.trc_name;
addq st elt, loc_comm_q, $1;
move #/l11/, st_elt.st_name;
335
move #/l_t11.traces/, st elt.trc.name;
addq st elt, loc.comm.q, #1;
move #/l_t12/, st elt.st.name;
move #/l_t12.traces/, st elt.trc.name;
addq st elt, loc.comm.q, #1;
move #/l_t13/, st elt.st.name;
move #/l_t13.traces/, st elt.trc.name;
addq st elt, loc.comm.q, #1;
move #/l_t14/, st elt.st.name;
move #/l_t14.traces/, st elt.trc.name;
addq st elt, loc.comm.q, #1;
move #/l_t15/, st elt.st.name;
move #/l_t15.traces/, st elt.trc.name;
addq st elt, loc.comm.q, #1;
move #/l_t16/, st elt.st.name;
move #/l_t16.traces/, st elt.trc.name;
addq st elt, loc.comm.q, #1;

if (mod_name = #/nodes/);

look;

move #/sh_t1/, st elt.st.name;
movc #/sh_t1.trcs/, st elt.trc.name;
addq st elt, comm.q, #1;
move #/sh_t2/, st elt.st.name;
movc #/sh_t2.trcs/, st elt.trc.name;
addq st elt, comm.q, #1;
move #/sh_t3/, st elt.st.name;
movc #/sh_t3.trcs/, st elt.trc.name;
addq st elt, comm.q, #1;
move #/sh_t4/, st elt.st.name;
movc #/sh_t4.trcs/, st elt.trc.name;
addq st elt, comm.q, #1;
move #/sh_t5/, st elt.st.name;
movc #/sh_t5.trcs/, st elt.trc.name;
addq st elt, comm.q, #1;
move #/sh_t6/, st elt.st.name;
movc #/sh_t6.trcs/, st elt.trc.name;
addq st elt, comm.q, #1;
move #/sh_t7/, st elt.st.name;
movc #/sh_t7.trcs/, st elt.trc.name;
addq st elt, comm.q, #1;
move #/sh_t8/, st elt.st.name;
movc #/sh_t8.trcs/, st elt.trc.name;
addq st elt, comm.q, #1;
move #/sh_t9/, st elt.st.name;
movc #/sh_t9.trcs/, st elt.trc.name;
addq st elt, comm.q, #1;
move #/sh_t10/, st elt.st.name;
movc #/sh_t10.trcs/, st elt.trc.name;
addq st elt, comm.q, #1;
move #/sh_t11/, st elt.st.name;
movc #/sh_t11.trcs/, st elt.trc.name;
addq st elt, comm.q, #1;
move #/sh_t12/, st elt.st.name;
movc #/sh_t12.trcs/, st elt.trc.name;
addq st elt, comm.q, #1;
move #/sh_t13/, st elt.st.name;
movc #/sh_t13.trcs/, st elt.trc.name;
addq st elt, comm.q, #1;
move #/sh_t14/, st elt.st.name;
movc #/sh_t14.trcs/, st elt.trc.name;
addq st elt, comm.q, #1;
move #/sh_t15/, st elt.st.name;
movc #/sh_t15.trcs/, st elt.trc.name;
addq st elt, comm.q, #1;
move #/sh_t16/, st elt.st.name;
movc #/sh_t16.trcs/, st elt.trc.name;
addq st elt, comm.q, #1;
move #/sh_t17/, st elt.st.name;
movc #/sh_t17.trcs/, st elt.trc.name;
addq st elt, comm.q, #1;

unlock;
endif;

mov tasklist, proclist;
sizeq proclist, br_chck ;
mov curr_task, tempstr;
mov $1, st_chck ;
while br_chck <> $0;
delq proclist, tcb_name;
cxsw tcb_name;
*time curr_tcb.stat.begin;
move wt_chk, curr_tcb.num;
add 1, wt_chk, wt_chk;
if (tcb_name <> #/idle/);
lock;
*type nod_name;
delq loc_comm_q, st_elt;
move tcb_name, st_elt.nom;
*type #/--LOCAL-- TASK: /;
*type tcb_name;
*type #/ QUEUE: /;
*typecr st_elt.st_name;
addq st_elt, temp_q, #1;
unlock;
lock;
delq comm_q, st_elt;
while (st_elt.nom <> nil_string);
addq st_elt, comm_q, #1;
delq comm_q, st_elt;
endwhile;
move tcb_name, st_elt.nom;
*type #/GLOBAL-- TASK: /;
*type tcb_name;
*type #/ QUEUE: /;
*typecr st_elt.st_name;
addq st_elt, comm_q, #1;
unlock;
endif;
move taskstart, curr_tcb.pcnt;
move tcb_name, curr_tcb.name;
move #/ready/, curr_tcb.status;
move curr_prio, curr_tcb.priority;
move taskset, curr_tcb.set;
addq curr_tcb, ttr, curr_prio;
sizeq proclist, br_chk;
endwhile;
move temp_q, loc_comm_q;
move nil_queue, temp_q;
crezw tempstr;
move #/rdy_tm/, rdy_str;
move #/wt_tm/, wt_str;
move #/run_tm/, run_str;
*move #/END RESPONSE ............/, end_str;
bnd evt_synch, #0;
bnd evt_loss_synch, #1;
bnd evt_msg_rx, #2;
bnd evt_signal, #3;
bnd evt_clk, #4;
enable W10;
call run_rtr_task;
ENDTASK;
ENDKERNEL;
B.2.3 Application Tasks

| Low priority idle task code |
| STAT rdy_tm, integer; |
| STAT ut_tm, integer; |
| STAT run_tm, integer; |

BEGIN
CALLP nilprim;
SETSTATE #/start/;
LOOP;
STATSELECT;
STATE start;
IF true;
CALLP nilprim;
SETSTATE #/start/;
ENDIF;
ENDSTATE;
ENDSELECT;
ENDLOOP
ENDTASK;

M30 Task

This task services the message interrupt and performs message transfer to and from the collector node.

VAR byte_cnt, integer; No. of message bytes transmitted or received
VAR msg_pending, integer;
VAR msg_out, t_msg_out;
VAR msg_in, t_msg_in;
VAR task_strt, integer;
VAR task_end, integer;
STAT cycle, integer;

ACTIVITY init;
  *time task_strt;
  move $0, msg_pending;
  move $0, msg_in.type;
ENDACT;

ACTIVITY upd_cycle;
  *time task_end;
  *sub task_end, task_strt, cycle;
  *time task_strt;
ENDACT;

ACTIVITY int_cnt;
  move $0, byte_cnt;
  move $0, msg_rdy;
ENDACT;

ACTIVITY incr_cnt;
  printer #/incr count in mag task/;
  add $1, byte_cnt, byte_cnt;
ENDACT;

ACTIVITY send_tone;
  move $2, msg_out.type;
  move #/mag, msg_out.src;
ENDACT;

ACTIVITY send_start;
  move $1, msg_out.type;
  move #/mag, msg_out.src;
ENDACT;

ACTIVITY send_diag;
  move $3, msg_out.type;
  move #/mag, msg_out.src;
ENDACT;

ACTIVITY ack_in;
  if (msg_in.type <> $0);
    move $1, msg_pending;
    endif;
ENDACT;

ACTIVITY clr_msg;
  move $0, msg_in.type;
  move $0, msg_pending;
ENDACT;

ACTIVITY set_sentry;
  ticks $200;
  move $0, msg_in.type;
  move $0, msg_pending;
ENDACT;

BRUTE
DO.ACT init;
CALLP nilprim;
SETSTATE #/clear/;
LOOP;
STATSELECT;
STATE clear;
IF true;
DO.ACT upd_cycle;
CALLP noresp;
SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
IF (msg_pending = $1);
CALLP nilprim;
SETSTATE #/hd1_msg/;
ENDIF;
IF (msg_rdy = $1);
DO.ACT init_cnt;
CALLP nilprim;
SETSTATE #/rxtr/;
ENDIF;
IF true;
CALLP norev, $0, #/msg_in/;
SETSTATE #/sigmsg/;
ENDIF;
ENDIF;
ENDSTATE;

STATE sigmsg;
IF true;
DO.ACT ack_in;
CALLP nilprim;
SETSTATE #/start/;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDSTATE;

STATE rxtr;
IF (byte_cnt < $32);
DO.ACT incr_cnt;
CALLP nilprim;
SETSTATE #/rxtr/;
ENDIF;
IF true;
CALLP nilprim;
SETSTATE #/mag_fin/;
ENDIF;
ENDIF;
ENDSTATE;

STATE mag_fin;
IF (msgtype = $1);
DO.ACT send_start;
CALLP send, #/init/, #/msg_out/;
SETSTATE #/clear/;
ENDIF;
IF (msgtype = $2);
DO.ACT send_tone;
CALLP send, #/tones/, #/msg_out/;
SETSTATE #/clear/;
ENDIF;
IF (msgtype = #3);
DO_ACT send_diag;
CALLP send, #/diags/, #/msg_out/;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;
STATE hndl_msg;
IF (msg_in.type = $7);
DO_ACT clr_msg;
CALLP endresp, $1;
SETSTATE #/clear/;
ENDIF;
STATE hndl_chnl;
IF (msg_in.type = $6);
DO_ACT set_synch;
CALLP endresp, $2;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;
ENDESELECT;
ENDBUDD
ENDTASK;

SYNCH Task
This task reports synch faults to the FAULT task and changes synch sources about every 1000 time units by sending a message to the MSG task.

STAT rdy_tm, integer;
STAT ut_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_alt;
VAR task_strt, integer;
VAR task_end, integer;
STAT cycle, integer;

ACTIVITY init;
  *time task_strt;
  move $0, synch_loss;
ENDACT;

ACTIVITY upd_cycle;
  *time task_end;
  sub task_end, task_strt, cycle;
  *time task_strt;
ENDACT;

ACTIVITY report_synch;
  move $0, synch_loss;
  move $9, msg_out.type;
  move #/synch/, msg_out.arc;
ENDACT;

ACTIVITY change_synch;
  move $0, msg_out.type;
  move #/synch/, msg_out.arc;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
ENDACT;

CONNECT Task
This task services requests for connections by simulating a fixed time delay. If the request is from the switching center, the tone request response is completed.

STAT rdy_tm, integer;
STAT ut_tm, integer;
STAT run_tm, integer;
VAR msg_in, t_msg_alt;
VAR task_strt, integer;
VAR task_end, integer;
STAT cycle, integer;

ACTIVITY init;
  *time task_strt;
ENDACT;

ACTIVITY upd_cycle;
  *time task_end;
  sub task_end, task_strt, cycle;
  *time task_strt;
ENDACT;

ACTIVITY make_connn;
  ticks $600;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
ENDACT;
STATE got_msg;
IF ((msg_in.src = #tune1) &
(msg_in.data = #1));
DO_ACT make_conn;
CALLP endresp, #3;
SETSTATE #/clear/;
ENDIF;

IF #true;
DO_ACT make_conn;
CALLP nilprim;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;
ENDSELECT;
ENDLOOP
ENDTASK;

CEHLS Task

This task services requests for channel
allocations and deallocations.
It maintains only 16 channels. When an
allocation request is
made, the requestor must wait for an
acknowledgement message.

STAT rdy_tm, integer;
STAT ut_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_elts;
VAR msg_in, t_msg_elts;
VAR num_allocated, integer;
VAR tsk_strt, integer;
VAR tsk_end, integer;
VAR cycle, integer;
VAR mtoch, Integer;
STAT cph, integer;
STAT over_limit, integer;

ACTIVITY initial;
+time tsk_strt;
ENDACT;

ACTIVITY upd_cycle;
+time tsk_end;
+sub tsk_end, tsk_strt, cycle;
+time tsk_strt;
ENDACT;

ACTIVITY get_chnl;
add #1, num_allocated, num_allocated;
+if (msg_in.src = #diag); rudd #1, mtoch, mtoch;
+else;
add #1, cph, cph;
+endif;
move #14, msg_out.type;
movle #/chnls/, msg_out.src;
ENDACT;

ACTIVITY ovtflx;
+add #1, over_limit;
ENDACT;

ACTIVITY free_chnl;
+sub num_allocated, #1, num_allocated;
+if (num_allocated < #0);
movle #0, num_allocated;
+endif;
ENDACT;

BEGIN
DO.ACT initial;
CALLP nilprim;
SETSTATE #/clear/;

LOC:
STATESELECT;
STATE clear;
IF #true;
DO.ACT upd_cycle;
CALLP neresp;
SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
IF (num_allocated < #15);
CALLP rec_in, #0, #/msg_in/;
SETSTATE #/got_msg/;
ENDIF;

IF #true;
DO.ACT ovtflx;
CALLP rec_in, #1;
SETSTATE #/free_req/;
ENDIF;
ENDSTATE;

STATE got_msg;
IF (msg_in.type = #12);
DO.ACT get_chnl;
CALLP send, msg_in.src, #/msg_out/;
SETSTATE #/clear/;
ENDIF;

IF #true;
DO.ACT free_chnl;
CALLP nilprim;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

STATE free_req;
IF #true;
DO.ACT free_chnl;
CALLP nilprim;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

ENDSELECT;
ENDLOOP
ENDTASK;

CINIT Task

This task accepts messages from the MSG
and to setup a call. It
doing, the CINIT task requests a channel,
connects it, and informs
the STAT task that the call is in progress.

STAT rdy_tm, integer;
STAT ut_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_elts;
VAR msg_in, t_msg_elts;
VAR tsk_strt, integer;
VAR tsk_end, integer;
STAT cycle, integer;

ACTIVITY init;
+time tsk_strt;
ENDACT;

ACTIVITY upd_cycle;
+time tsk_end;
+sub tsk_end, tsk_strt, cycle;
+time tsk_strt;
ENDACT;

ACTIVITY chnl.req;
movle #12, msg_out.type;
movle #/cinit/, msg_out.src;
ENDACT;

ACTIVITY conn_req;
move $a1, msg_out.type);
move $/cinit/, msg_out.src;
ENDACT;

ACTIVITY stat_req;
move $a16, msg_out.type);
move $/cinit/, msg_out.src;
ENDACT;

BEGIN
DU.ACT init;
CALLP nilprim;
SETSTATE #/clear/;

; LOOP
; STATESSELECT:

; STATE clean;
; IF #true;
; DU.ACT upd_cycle;
; CALLP neuresp;
; SETSTATE #/strt/;
; ENDIF;
; ENDSTATE;

; STATE start;
; IF #true;
; CALLP recv, $a1, #/msg_in/;
; SETSTATE #/got_msg/;
; ENDIF;
; ENDSTATE;

; STATE got_msg;
; IF #true;
; DU.ACT chnl_req;
; CALLP send, #/chnl/, #/msg_out/;
; SETSTATE #/wt_ack/;
; ENDIF;
; ENDSTATE;

; STATE wt_ack;
; IF #true;
; CALLP recv, $a16, #/msg_in/;
; SETSTATE #/got_chnl/;
; ENDIF;
; ENDSTATE;

; STATE got_chnl;
; IF #true;
; DU.ACT conn_req;
; CALLP send, #/connect/, #/msg_out/;
; SETSTATE #/got_conn/;
; ENDIF;
; ENDSTATE;

; STATE got_conn;
; IF #true;
; DU.ACT stat_req;
; CALLP send, $/stat/, #/msg_out/;
; SETSTATE #/clear/;
; ENDIF;
; ENDSTATE;

; ENDSELECT;
; ENDLOOP
; ENDTASK;

; STAT Task
; This task accepts reports of call setup
; completions. In response,
; it requests that the SIGNALLING task look
; for call teardown messages.
; STAT also accepts reports of integrity
; failures from the
; SIGNALLING task. When this happens, the
; report is sent to the
; FAULT task. Finally messages from the EBD

; STAT rdy_tm, integer;
; STAT wt_tm, integer;
; STAT run_tm, integer;
; VAR msg_out, t_msg_out;
; VAR msg_in, t_msg_in;
; VAR tak_strt, integer;
; VAR tak_end, integer;
; STAT cycle, integer;

ACTIVITY init;
*time tak_strt;
ENDACT;

ACTIVITY upd_cycle;
*time tak_end;
*sub tak_end, tak_strt, cycle;
*time tak_strt;
ENDACT;

ACTIVITY new_call;
ticks #200;
update table
move $a1, msg_data;
move $a16, msg_out.type;
move $/stat/, msg_out.src;
ENDACT;

ACTIVITY call_fin;
ticks #200;
update table
move $a1, msg_data;
ENDACT;

ACTIVITY integ_fail;
move $a10, msg_out.type;
move $a1, msg_data;
move $/stat/, msg_out.src;
ENDACT;

BEGIN
DU.ACT init;
CALLP nilprim;
SETSTATE #/clear/;

; LOOP
; STATESSELECT:

; STATE clean;
; IF #true;
; DU.ACT upd_cycle;
; CALLP neuresp;
; SETSTATE #/strt/;
; ENDIF;
; ENDSTATE;

; STATE start;
; IF #true;
; CALLP recv, $a0, #/msg_in/;
; SETSTATE #/got_msg/;
; ENDIF;
; ENDSTATE;

STATE got_msg;
IF (msg_in.type == $16);
DU.ACT new-call;
CALLP send, #/signal/, #/msg_out/;
SETSTATE #/response/;
ENDIF;
IF (msg_in.type == $17);
DU.ACT integ-fail;
CALLP send, #/fault/, #/msg_out/;
SETSTATE #/clear/;
ENDIF;
IF #true;
DU.ACT call_fin;
CALLP nilprim;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;
STATE response;
IF $true;
    CALLP endresp, #5;
ENDIF;
ENDSTATE;
ENDSELECT;
ENDLOOP
ENDTASK;

END Task
This task accepts messages from the
SIGNALLING task reporting
that a caller has hung up. This causes the
END task to request a
disconnection and the freeing of the
channel. It completes by
informing the STAT task that the takedown
is complete.

STAT rdy_tm, integer;
STAT st_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_elt;
VAR msg_in, t_msg_elt;
VAR tskstrt, integer;
VAR tsk_end, integer;
STAT cycle, integer;

ACTIVITY init;
   $time tskstrt;
ENDACT;

ACTIVITY upd_cycle;
   $time tsk_end;
   $sub tsk_end, tskstrt, cycle;
   $time tskstrt;
ENDACT;

ACTIVITY chnl_req;
   move #11, msg_out.type;
   move #/end/, msg_out.src;
ENDACT;

ACTIVITY comm_req;
   move #11, msg_out.type;
   move #/end/, msg_out.src;
ENDACT;

ACTIVITY stat_req;
   move #19, msg_out.type;
   move #/end/, msg_out.src;
ENDACT;

BEGIN
DO ACT init;
    CALLP nilprim;
    SETSTATE #/clear/;
END;

BEGIN
STATESELECT;
    STATE clear;
    IF $true;
        DO ACT
            CALLP
            newresp;
            SETSTATE #/start/;
    ENDIF;
    ENDSTATE;

    STATE start;
    IF $true;
        CALLP
        recv, #/msg_in/;
    SETSTATE #/got_msg/;
    ENDIF;
ENDLOOP
ENDTASK;

STATE got_msg;
IF $true;
    DO ACT
        comm_req;
        CALLP send, #/connect/, #/msg_out/;
        SETSTATE #/rel_chnl/;
    ENDIF;
ENDSTATE;

STATE rel_chnl;
IF $true;
    DO ACT
        chnl_req;
        CALLP send, #/chnls/, #/msg_out/;
        SETSTATE #/call_fin/;
    ENDIF;
ENDSTATE;

STATE call_fin;
IF $true;
    DO ACT
        stat_req;
        CALLP send, #/stat/, #/msg_out/;
        SETSTATE #/clear/;
    ENDIF;
ENDSTATE;
ENDSELECT;
ENDLOOP
ENDTASK;

FAULT Task
This task accepts reports of faults and
sends them to the MSG
task for transmission. If a fault is
received from the TMTC or
STAT tasks, then 1% of data must be
transferred between these tasks.

STAT rdy_tm, integer;
STAT st_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_elt;
VAR msg_in, t_msg_elt;
VAR tskstrt, integer;
VAR tsk_end, integer;
STAT cycle, integer;

ACTIVITY init;
   $time tskstrt;
ENDACT;

ACTIVITY upd_cycle;
   $time tsk_end;
   $sub tsk_end, tskstrt, cycle;
   $time tskstrt;
ENDACT;

ACTIVITY chnl_req;
   move #11, msg_out.type;
   move #/end/, msg_out.src;
ENDACT;

ACTIVITY comm_req;
   move #11, msg_out.type;
   move #/end/, msg_out.src;
ENDACT;

ACTIVITY stat_req;
   move #19, msg_out.type;
   move #/end/, msg_out.src;
ENDACT;

ACTIVITY send_report;
   move #7, msg_out.type;
   move #/fault/, msg_out.src;
ENDACT;

BEGIN
DO ACT init;
    CALLP nilprim;
    SETSTATE #/clear/;
END;

BEGIN
STATESELECT;
    STATE clear;
    IF $true;
        DO ACT
            upd_cycle;
            CALLP
            newresp;
            SETSTATE #/start/;
    ENDIF;
    ENDSTATE;

    STATE start;
    IF $true;
        CALLP
        recv, #/msg_in/;
    SETSTATE #/got_msg/;
    ENDIF;
ENDLOOP
ENDTASK;
STATE start;
  IF $true;
    CALLP recv, 0, #/msg_in/;
ENDIF;
ENDSTATE;

STATE get_msg;
  IF (msg_in.src = #/time/);
    CALLP get_data, msg_in.src, #/vi_data/, #/1000/;
    SETSTATE #/report/;
  ENDIF;
ENDSTATE;

STATE report;
  IF $true;
    CALLP nilprim;
  ENDIF;
ENDSTATE;

STATE select_tone;
ENDSTATE;

STATE clear;
  IF $true;
    CALLP send, #/connect/, #/msg_out/;
  ENDIF;
ENDSTATE;

BEGIN
  DO ACT init;
    CALLP nilprim;
    SETSTATE #/clear/;
  END;
ENDTASK;

LOOP;
STATESELECT;
  STATE clear;
    IF $true;
      CALLP send, #/connect/, #/msg_out/;
    ENDIF;
ENDSTATE;

STATE report;
    CALLP send, #/connect/, #/msg_out/;
  ENDIF;
ENDSTATE;

STATE select_tone;
    CALLP select_tone;
ENDSTATE;

STATE clear;
    CALLP send, #/connect/, #/msg_out/;
ENDSTATE;

SIGNAL Task
This task collects messages from the signalling hardware every second by way of an interrupt. The message can be: integrity lost, integrity okay, or call takedown. Nothing is done for integrity okay. For integrity failure, the STAT task is informed and a request is sent to the DIAU task. For call takedown, the STAT task is informed. Messages are also accepted to start and stop integrity monitoring. These actions are simulated by a fixed delay.

STATE rdy_tm, integer;
STATE wtm_tm, integer;
STATE run_tm, integer;
VAR msg_out, tmsg_elit;
VAR msg_in, tmsg_elit;
VAR tast_end, integer;
VAR tast_start, integer;
STATE cycle, integer;

ACTIVITY init;
  time tast_start;
ENDACT;

ACTIVITY upd_cycle;
  time tast_end;
  *time tast_start, tast_end, cycle;
ENDACT;

ACTIVITY select_tone;
  time #360;
  if (msg_in.src = #/msg/); move 0, msg_out.data;
  else;
    move 0, msg_out.data;
  endif;
  move #11, msg_out.type;
  move #/tone/, msg_out.src;
ENDACT;

BEGIN
  DO ACT init;
    CALLP nilprim;
    SETSTATE #/clear/;
  END;
ENDTASK;
AUDITS Task

This is a periodic task that requests diagnostics from the DIAGS task whenever the time for one of the three audits comes up. A clock handling primitive provides a time base for the task.

STAT rdy_tm, integer;
STAT ut_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_elt;
VAR audit_cnt, integer;
VAR und2_cnt, integer;
VAR und3_cnt, integer;
VAR task_strt, integer;
VAR task_end, integer;
STAT cycle, integer;

ACTIVITY Init;
**time task_strt;
ENDACT;

ACTIVITY upd_cycle;
**time task_end;
**sub task_end, task_strt, cycle;
**time task_strt;
ENDACT;

ACTIVITY incr_cnt;
add $1, audit_cnt, audit_cnt;
add $1, und2_cnt, und2_cnt;
add $1, und3_cnt, und3_cnt;
ENDACT;

ACTIVITY tell_tm;
move $23, msg_out.type;
move #/audits/, msg_out.src;
ENDACT;

ACTIVITY aud1;
move $0, audit_cnt;
move $10, msg_out.type;
move #/audits/, msg_out.src;
ENDACT;

ACTIVITY aud2;
move $0, und2_cnt;
move $10, msg_out.type;
move #/audits/, msg_out.src;
ENDACT;

ACTIVITY aud3;
move $0, und3_cnt;
move $10, msg_out.type;
move #/audits/, msg_out.src;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
LOOP;
STATESELECT;
STATE clear;
IF true;
DO_ACT upd_cycle;
CALLP newresp;
SETSTATE #/start/;
ENDIF;
ENDSTATE;
STATE start;
IF true;
DO_ACT incr_cnt;
CALLP sleep;
ENDSTATE;
STATE trigger_tm;
IF true;
DO_ACT tell_tm;
CALLP send, #/time/, #/msg_out/;
SETSTATE #/chk_aud1/;
ENDIF;
ENDSTATE;
STATE chk_aud1;
IF (aud1_cnt = $10);
DO_ACT aud1;
CALLP send, #/diags/, #/msg_out/;
SETSTATE #/chk_aud2/;
ENDIF;
ENDSTATE;
STATE chk_aud2;
IF (aud2_cnt = $20);
DO_ACT aud2;
CALLP send, #/diags/, #/msg_out/;
SETSTATE #/chk_aud3/;
ENDIF;
ENDSTATE;
STATE chk_aud3;
IF (aud3_cnt = $30);
DO_ACT aud3;
CALLP send, #/diags/, #/msg_out/;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;
ENDSELECT;
ENDLOOP
ENDTASK;

DIAGS Task

This task accepts messages to run diagnostics from the MSG task, AUDITS task, and the SIGNAL task. When such a request is received, two channels are requested from the CHRLS task and then a message is sent to the appropriate diagnostic task.

STAT rdy_tm, integer;
STAT ut_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_elt;
VAR msg_in, t_msg_elt;
VAR dest, string;
VAR dg_num, integer;
VAR task_strt, integer;
VAR task_end, integer;
STAT cycle, integer;

ACTIVITY Init;
**time task_strt;
ENDACT;

ACTIVITY upd_cycle;
**time task_end;
sub task_end, task_start, cycle;
    time task_start;
ENDACT;
ACTIVITY chnl_req;
    move #12, msg_out.type;
    move #diafg, #msg_out.ssrc;
ENDACT;
ACTIVITY choose_diag;
    read #4, dg_num;
    move #diag, msg_out.type;
    if (dg_num = #0);
        move #tonedalg, dest;
    else;
        if (dg_num = #1);
            move #chd1alg, dest;
        else;
            move #chd2alg, dest;
        endif;
    endif;
ENDACT;
ENDACT;
DO.ACT init;
    CALLP nilprim;
    SETUPSTATE #clear/;
ENDACT;
LOOP;
    STATESSELECT;
    STATE clear;
    IF #true;
        DO.ACT upd_cycle;
        CALLP newsesp;
        SETUPSTATE #/start/;
    ENDIF;
    ENDSTATE;
    STATE start;
    IF #true;
        CALLP recv, #3, #/msg_in/;
        SETUPSTATE #/got_msg/;
    ENDIF;
    ENDSTATE;
    STATE get_msg;
    IF #true;
        DO.ACT chnl_req;
        CALLP send, #chnl3, #/msg_out/;
        SETUPSTATE #/st_ack1/;
    ENDIF;
    ENDSTATE;
    STATE st_ack1;
    IF #true;
        CALLP recv, #14, #/msg_in/;
        SETUPSTATE #/got_chnl/;
    ENDIF;
    ENDSTATE;
    STATE get_chnl;
    IF #true;
        DO.ACT chnl_req;
        CALLP send, #chnl3, #/msg_out/;
        SETUPSTATE #/st_ack3/;
    ENDIF;
    ENDSTATE;
    STATE st_ack3;
    IF #true;
        CALLP recv, #14, #/msg_in/;
        SETUPSTATE #/pick_diag/;
    ENDIF;
    ENDSTATE;
    STATE pick_diag;
    IF #true;
        DO.ACT choose_diag;
        CALLP send, dest, #/msg_out/;
        SETUPSTATE #/clear/;
    ENDIF;
    ENDSTATE;
    ENDSELECT;
ENDLOOP;

} TONEDIAG Task
This task services requests for tone
 diagnostics by sending a
 request message to the TONES task and
 then simulating the diagnostic
 by a fixed time delay.

VAR msg_out, t_msg_out;
VAR msg_in, t_msg_in;
VAR result, integer;
VAR fail, integer;
VAR task_strt, integer;
VAR task_end, integer;
VAR cycle, integer;

ACTIVITY init;
    time task_start;
ENDACT;

ACTIVITY upd_cycle;
    time task_end;
    sub task_end, task_strt, cycle;
    time task_strt;
ENDACT;

ACTIVITY select_tone;
    move #2, msg_out.type;
    move #tonedalg, msg_out.type;
ENDACT;

ACTIVITY do_diag;
    move #0, fail;
    tick #300;
    read #10, result;
    if (result = #0);
        move #1, fail;
    endif;
ENDACT;

ACTIVITY report_fail;
    move #10, msg_out.type;
    move #tonedalg, msg_out.type;
ENDACT;

BEGIN
    DO.ACT init;
    CALLP nilprim;
    SETUPSTATE #/clear/;
ENDACT;
LOOP;
    STATESSELECT;
    STATE clear;
    IF #true;
        DO.ACT upd_cycle;
        CALLP newsesp;
        SETUPSTATE #/start/;
    ENDIF;
    ENDSTATE;
    STATE start;
    IF #true;
        CALLP recv, #0, #/msg_in/;
        SETUPSTATE #/got_msg/;
### CHLDIAIG Task

This task services requests for channel diagnostics by sending a request message to the CONNECT task and then one to the SIGNAL task and finally simulating the diagnostic box by a fixed time delay.

```plaintext
STAT rdv_tn, integer;
STAT st_tn, integer;
STAT run_tn, integer;
VAR msg_out, t.msg_alt;
VAR msg_in, t.msg.out;
VAR result, integer;
VAR fail, integer;
VAR tsk_strt, integer;
VAR tsk_end, integer;
STAT cycle, integer;

ACTIVITY init;
  *time tsk_strt;
ENDACT;

ACTIVITY upd_cycle;
  *time tsk_end;
  *sub tsk_end, tsk_strt, cycle;
  *time tsk_strt;
ENDACT;

ACTIVITY connect_chnl;
  move $11, msg_out.type;
  move #/chldiaig/. msg_out.src;
ENDACT;

ACTIVITY stop_integ;
  move #20, msg_out.type;
  move #/chldiaig/. msg_out.src;
ENDACT;

ACTIVITY do_diag;
  move #30, fail;
ticks $180;
rand #10, result;
if (result = #9)
  move #1, fail;
ENDACT;

ACTIVITY report_fail;
  move #10, msg_out.type;
  move #/chldiaig/. msg_out.src;
ENDACT;

BEGIN
  DO_ACT init;
  CALLP nilprim;
  SETSTATE #/clear/;
  LOOP:
  STATESSELECT:
    STATE clear;
      IF true;
        DO_ACT upd_cycle;
        CALLP neuronp;
        SETSTATE #/start/;
      ENDIF;
      ENDSTATE;
    STATE start;
      IF true;
        CALLP recv, #0, #/msg_in/;
        SETSTATE #/get_msg/;
      ENDIF;
      ENDSTATE;
    STATE get_msg;
      IF true;
        DO_ACT connect_chnl;
        CALLP send, #/connect/, #/msg_out/;
        SETSTATE #/signal/;
      ENDIF;
      ENDSTATE;
    STATE signal;
      IF true;
        DO_ACT stop_integ;
        CALLP send, #/signal/, #/msg_out/;
        SETSTATE #/diagnose/;
      ENDIF;
      ENDSTATE;
    STATE diagnose;
      IF true;
        DO_ACT do_diag;
        CALLP nilprim;
        SETSTATE #/responses/;
      ENDIF;
      ENDSTATE;
    STATE chk_results;
      IF (fail = #1);
        DO_ACT report_fail;
        CALLP send, #/fault/, #/msg_out/;
        SETSTATE #/chk_results/;
      ENDIF;
      ENDSTATE;
  ENDSTATE;
```

```plaintext
STATE diagnose;
  IF true;
    DO_ACT do_diag;
    CALLP nilprim;
    SETSTATE #/responses/;
  ENDIF;
  ENDSTATE;

STATE chk_results;
  IF (fail = #1);
    DO_ACT report_fail;
    CALLP send, #/fault/, #/msg_out/;
    SETSTATE #/responses/;
  ENDIF;
  ENDSTATE;

STATE responses;
  IF (fail = #0);
    CALLP endresp, #6;
    SETSTATE #/clear/;
  ENDIF;
  ENDSTATE;

ENDSELECT;
ENDLOOP
ENDSTATE;
```
MSGDIAG Task

This task services requests for messaging diagnostics by sending a request message to the SIGNAL task and then simulating the diagnostic by a fixed time delay.

STAT rdy_tm, integer;
STAT ut_tm, integer;
STAT run_tm, integer;

VAR msg_out, t_msg_out;
VAR msg_in, t_msg_out;
VAR result, integer;
VAR fail, integer;

VAR tskstrt, integer;
VAR tskend, integer;
STAT cycle, integer;

ACTIVITY init;
  time tskstrt;
ENDACT;

ACTIVITY upd_cycle;
  time tskend;
  sub tskend, tskstrt, cycle;
  time tskstrt;
ENDACT;

ACTIVITY stop_integ;
  move $20, msg_out.type;
  move $/msgdiag/, msg_out.src;
ENDACT;

ACTIVITY do_diag;
  move $0, fail;
  ticks $200;
  rand $10, result;
  if (result = $0)
    move $1, fail;
ENDACT;

ACTIVITY report_fail;
  move $10, msg_out.type;
  move $/msgdiag/, msg_out.src;
ENDACT;

BEGIN
  DO.ACT init;
  CALLP nilprim;
  SETSTATE $/clear/;
END;

STATESELECT;

STATE clear;
  if $true
    DO.ACT
    CALLP
    SETSTATE $/start/;
END;
ENDSTATE;

STATE start;
  if $true
    CALLP recv, $0, $/msg_in/;
END;

STATE got_msg;
  if $true
    DO.ACT
    CALLP send, $/signal/,
    SETSTATE $/diagpose/;
END;
ENDSTATE;

STATE diagnose;
  if $true
    DO.ACT
    CALLP nilprim;
    SETSTATE $/chk_results/;
END;
ENDSTATE;

STATE chk_results;
  if (fail = $1)
    DO.ACT
    CALLP send, $/feedback/,
    SETSTATE $/response/;
END;
ENDSTATE;

STATE response;
  if (fail = $0)
    CALLP endresp, $0;
    SETSTATE $/clear/;
END;
ENDSTATE;

TIDIG Task

This task services requests for carrier diagnostics by sending a request message to the TINTC task and then simulating the diagnostic by a fixed time delay.

STAT rdy_tm, integer;
STAT ut_tm, integer;
STAT run_tm, integer;

VAR msg_out, t_msg_out;
VAR msg_in, t_msg_out;
VAR result, integer;
VAR fail, integer;

VAR tskstrt, integer;
VAR tskend, integer;
STAT cycle, integer;

ACTIVITY init;
  time tskstrt;
ENDACT;

ACTIVITY upd_cycle;
  time tskend;
  sub tskend, tskstrt, cycle;
  time tskstrt;
ENDACT;

ACTIVITY stop_integ;
  move $20, msg_out.type;
  move $/msgdiag/, msg_out.src;
ENDACT;

BEGIN
  DO.ACT init;
  CALLP nilprim;
  SETSTATE $/clear/;
END;

STATESELECT;

STATE clear;
  if $true
    DO.ACT
    CALLP
    SETSTATE $/start/;
END;
ENDSTATE;

STATE start;
  if $true
    CALLP recv, $0, $/msg_in/;
END;

STATE got_msg;
  if $true
    DO.ACT
    CALLP send, $/signal/,
    SETSTATE $/diagpose/;
END;
ENDSTATE;

STATE diagnose;
  if $true
    DO.ACT
    CALLP nilprim;
    SETSTATE $/chk_results/;
END;
ENDSTATE;

STATE chk_results;
  if (fail = $1)
    DO.ACT
    CALLP send, $/feedback/,
    SETSTATE $/response/;
END;
ENDSTATE;

STATE response;
  if (fail = $0)
    CALLP endresp, $0;
    SETSTATE $/clear/;
END;
ENDSTATE;

348
move #/tidiag/, msg_out.scr;
EINDACT;

ACTIVITY do_diag;
move #0, fail;
tick #400;
rand #10, result;
if (result = #0) move #1, fail;
end;
EINDACT;

ACTIVITY report.fail;
move #10, msg_out.type;
move #/tidiag/, msg_out.scr;
EINDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
END:

STATE SELECT:

STATE clear;
IF #true:
DO_ACT upd_cycle;
CALLP
SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
IF #true:
CALLP recv, #0, #/msg_in/;
SETSTATE #/got_msg/;
ENDIF;
ENDSTATE;

STATE got_msg;
IF #true:
DO_ACT stop_mtc;
CALLP send, #/signal/, #/msg_out/;
SETSTATE #/diagnost/;
ENDIF;
ENDSTATE;

STATE diagnosis;
IF #true:
DO_ACT do_diag;
CALLP nilprim;
SETSTATE #/chk_results/;
ENDIF;
ENDSTATE;

STATE chk_results;
IF (fail = #1):
DO_ACT report_fail;
CALLP send, #/fault/, #/msg_out/;
SETSTATE #/response/;
ENDIF;
IF #true:
CALLP nilprim;
SETSTATE #/response/;
ENDIF;
ENDSTATE;

STATE response;
IF (fail = #0):
CALLP endresp, #5;
SETSTATE #/clear/;
ENDIF;
IF #true:
CALLP nilprim;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

ENDSELECT:
ENDUP
ENDTASK;

TMTG Task

This task performs routine carrier maintenance every second (as signalled by the arrival of a message from the AUDIT task). It also accepts messages from TIDIAG to stop scanning carriers.

STAT rdy_tm, integer;
STAT wt_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg-alt;
VAR msg_in, t_msg-alt;
VAR result, integer;
VAR fail, integer;
VAR tak_strt, integer;
VAR tak_end, integer;
STAT cycle, integer;

ACTIVITY init;
setm tak_strt;
EINDACT;

ACTIVITY upd_cycle;
setm tak_end;
sub tak_end, tak_strt, cycle;
sub tak_strt, tak_strt;
EINDACT;

ACTIVITY do_scan;
move #0, fail;
move #1, ci_data;
tick #100;
rand #10, result;
if (result = #0) move #1, fail;
endif;
EINDACT;

ACTIVITY report_fail;
move #10, msg_out.type;
move #/timo/, msg_out.scr;
EINDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
END:

STATE SELECT:

STATE clear;
IF #true:
DO_ACT upd_cycle;
CALLP
SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
IF #true:
CALLP recv, #0, #/msg_in/;
SETSTATE #/got_msg/;
ENDIF;
ENDSTATE;

STATE got_msg;
IF #true:
DO_ACT stop_mtc;
CALLP send, #/signal/, #/msg_out/;
SETSTATE #/diagnost/;
ENDIF;
ENDSTATE;

STATE diagnosis;
IF #true:
DO_ACT do_diag;
CALLP nilprim;
SETSTATE #/chk_results/;
ENDIF;
ENDSTATE;

STATE chk_results;
IF (fail = #1):
DO_ACT report_fail;
CALLP send, #/fault/, #/msg_out/;
SETSTATE #/response/;
ENDIF;
IF #true:
CALLP nilprim;
SETSTATE #/response/;
ENDIF;
ENDSTATE;

STATE response;
IF (fail = #0):
CALLP endresp, #5;
SETSTATE #/clear/;
ENDIF;
IF #true:
CALLP nilprim;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

ENDSELECT:
ENDUP
ENDTASK;

349
STATE chk_results;
    IF (fail = #1);
      DO_ACT report_fail;
        CALLP send, #/fault/, #/msg_out/;
      SETSTATE #/response/;
    ENDIF;
  IF (true);
    CALLP nilprim;
    SETSTATE #/response/;
  ENDIF;
ENDSTATE;

STATE response;
  IF (fail = #0);
    CALLP endresp, #/;
    SETSTATE #/clear/;
  ENDIF;
  IF (true);
    CALLP nilprim;
    SETSTATE #/clear/;
  ENDIF;
ENDSTATE;
ENDESELECT;
ENDLOOP
ENDDATA;
B.3 IPC Selection Methods

The following Design-Aid code is that used to model the various kernels and applications used in the IPC section of the design method chapter. The kernel and application used for the case of nonblocking send with the blocking receive specific primitives is the same as that used to model the hardware platform. As such it will not be duplicated here.

B.3.1 Nonblocking Send With General Receive Kernel

The following kernel is identical to the nonblocking kernel of the hardware platform section except that the receive primitive has been modified so that receive specific types of requests are no longer supported.

```plaintext
KERNEL CODE FILE
This kernel contains the following primitives:
  nonblocking send
  general blocking receive
  sleep
  get data from another task
  nilprim

The following external message types are supported:
  #1 - start call setup
  #2 - provide tone
  #3 - run diagnostic
  #4 - integrity ok
  #6 - integrity failure
  #6 - call takedown

The following internal message types are supported:
  #1 - START - request call setup
  #2 - TONES - request for tone
  #3 - DIAG - request single random diagnostic
  #4 -
  #5 -
  #7 - MSG - report fault
  #8 - MSG - perform and report sinc operation
  #9 - FAULT - sinc loss
  #10 - FAULT - general fault report
  #11 - CONNECT - perform connect/disconnect
  #12 - CHRLS - allocate channel
  #13 - CHRLS - free channel
  #14 - all tasks - channel granted
  #15 - STAT - call setup complete
  #16 - SIGNAL - start integrity checking
  #17 - STAT - integrity failure report
  #18 - END - caller hung up
  #19 - STAT - call finished
  #20 - STAT - stop integrity checking
  #21 - all diag - run diagnostic
  #22 - TINTC - stop carrier checking
  #23 - TINTC - monitor carriers

The following response types are supported:
  #1 - MSG - fault report sent to switching center
  #2 - MSG - sinc source changed
  #3 - CONNECT - switching center tone request connected
  #4 - STAT - call finished
  #6 - STAT - call started
  #6 - all diags - diagnostic completed without failure
  #7 - TINTC - carrier scan completed without failure

KERNEL basek;
TASK kern_task, 50;
TYPE t_sock_state = rdy_time : integer *;
run_time : integer *;
vt_time : integer *;
begin : integer *;
finish : integer *;
resp_q : queue ; of t_trace_elt

TYPE t_shared_state = data_trace : queue ; of t_trace_elt
```

351
TYPE t_tcb = name : string * ; task name
num : integer * ;
status : string * ; task state
priority : integer * ; priority
pcnt : integer * ; program counter
elt_str : string * ; rcv msg elt
stk : stack * ; local task stack
sta : t_tcb_stats * ; task control block

TYPE t_sh_data = task_num : string * ; task name
num_msgs : integer * ; number of queued messages
msgs : queue * ; message queue (t_mag_elt)
trcs : queue * ; trace queue (t_trace_elt)

TYPE t_mag_struct = task_num : string * ; task name
waiting : integer * ; flag set if blocked for msg
msg_q : queue * ; message queue (t_mag_elt)
traces : queue * ; trace queue (t_trace_elt)
wt_type : integer * ; type of msg blocked on
block : t_tcb * ; tcb wait area

TYPE t_id_struct = nom : string * ;
tx_name : string * ;
at_name : string *

SHARED comm_q, queue * of t_id_struct (one elt per task);
SHARED sh_t1, t_sh_data *;
SHARED sh_t2, t_sh_data *;
SHARED sh_t3, t_sh_data *;
SHARED sh_t4, t_sh_data *;
SHARED sh_t5, t_sh_data *;
SHARED sh_t6, t_sh_data *;
SHARED sh_t7, t_sh_data *;
SHARED sh_t8, t_sh_data *;
SHARED sh_t9, t_sh_data *;
SHARED sh_t10, t_sh_data *;
SHARED sh_t11, t_sh_data *;
SHARED sh_t12, t_sh_data *;
SHARED sh_t13, t_sh_data *;
SHARED sh_t14, t_sh_data *;
SHARED sh_t15, t_sh_data *;
SHARED sh_t16, t_sh_data *;
SHARED sh_t17, t_sh_data *;
SHARED sh_t18, t_sh_data *;
SHARED sh_t19, t_sh_data *;
SHARED sh_t20, t_sh_data *;
SHARED sh_t21, t_sh_data *;
SHARED sh_t22, t_sh_data *;
SHARED sh_t23, t_sh_data *;
SHARED sh_t24, t_sh_data *;
SHARED sh_t25, t_sh_data *;
SHARED sh_t26, t_sh_data *;
SHARED sh_t27, t_sh_data *;
SHARED sh_t28, t_sh_data *;
SHARED sh_t29, t_sh_data *;
SHARED sh_t30, t_sh_data *;

; Message queues
VAR loc_comm_q, queue * of t_id_struct (one elt per local task)
VAR at_elt, t_id_struct *
VAR l_t1, t_mag_struct *
VAR l_t2, t_mag_struct *
VAR l_t3, t_mag_struct *
VAR l_t4, t_mag_struct *
VAR l_t5, t_mag_struct *
VAR l_t6, t_mag_struct *
VAR l_t7, t_mag_struct *
VAR l_t8, t_mag_struct *
VAR l_t9, t_mag_struct *
VAR l_t10, t_mag_struct *
VAR l_t11, t_mag_struct *
VAR l_t12, t_mag_struct *
VAR l_t13, t_mag_struct *
VAR l_t14, t_mag_struct *
VAR l_t15, t_mag_struct *
VAR l_t16, t_mag_struct *
VAR l_t17, t_mag_struct *
VAR l_t18, t_mag_struct *
VAR l_t19, t_mag_struct *
VAR l_t20, t_mag_struct *
VAR l_t21, t_mag_struct *
VAR l_t22, t_mag_struct *
VAR l_t23, t_mag_struct *
VAR l_t24, t_mag_struct *
VAR l_t25, t_mag_struct *
VAR l_t26, t_mag_struct *
VAR l_t27, t_mag_struct *
VAR l_t28, t_mag_struct *
VAR l_t29, t_mag_struct *
VAR l_t30, t_mag_struct *
VAR local, string *
VAR shared, string *
VAR loc_tdc, string *
VAR sh_tdc, string *

STAT sends, integer *
STAT os_cvh, integer *
STAT switches, integer *
STAT nn_send, integer *
STAT nn_recv, integer *
STAT tt_send, integer *
STAT tt_recv, integer *
STAT wv0, integer *
STAT sv0, integer *
STAT sv1, integer *

352
STAT evt_2, integer;
STAT evt_3, integer;
STAT evt_4, integer;
VAR start_os, integer;
VAR end_os, integer;
VAR elapsed, integer;
VAR rdy_str, string;
VAR ws_str, string;
VAR run_str, string;
VAR end_str, string;
VAR wait, queue;
VAR wt_chk, integer;
VAR msg_str, string;
VAR dest_str, string;
VAR temp_tcb, t_tcb;
VAR temp2_tcb, t_tcb;
VAR ret_po, integer;
VAR tmp_sth, stack;
VAR num_parms, integer;
VAR found, integer;
VAR msg_type, integer;
VAR data_size, integer;
VAR nil_queue, queue;
VAR nil_string, string;
VAR msg_alt, t_msg_alt;
VAR m_num, integer;
VAR temp2_q, queue;
VAR msg_type, integer;
VAR q_num, integer;
VAR done, integer;

; Global Kernel Variables
VAR curr_tcb, t_tcb ; current task control block
VAR proclist, queue ; copy of task name list
VAR br_cchk, integer ; branch condition check
VAR tempstr, string ; temporary string value
VAR tcb_name, string ; current task name
VAR rtr, queue ; ready to run queue

; Variables for get_local and get_sh procedures
VAR g_q_size, integer;
VAR g_found, integer;
VAR g_st elt, t_id strct;

; VARIABLE declarations for procedure RUN_RTR_TASK
VAR error, boolean; Set if no ready processes
VAR rtr_pcb, t_tcb; FQ3 to be scheduled

; PROCEDURE run_rtr_task
; This procedure takes the first ready to run task off the rtr
; queue (this is the one with the highest priority) and copies
; its saved state into the current processor state. If there are
; no ready to run tasks, then an error message is printed and
; the kernel terminates.
PROC run_rtr_task;
sizeq xtr, br_cchk;
IF (br_cchk = 0);
printerr &/ERROR: NO RTR PROCESSES/;
halt;
ELSE;
delq rtr, curr_tcb;
move curr_tcb.name, tcb_name;
extern tcb_name;
add $1, switches, switches;
@time curr_tcb.sts.finish;
@sub curr_tcb.sts.finish, curr_tcb.sts.begin, curr_tcb.sts.begin;
add curr_tcb.sts.begin, curr_tcb.sts.rdy_time, curr_tcb.sts.rdy_time;
move $/rdy_tm/, rdy_str;
move curr_tcb.sts.rdy_time, @rdy_str;
move curr_tcb.sts.et_time, @et_str;
@time curr_tcb.sts.begin;

move #/running/, curr_tcb.status;
movw curr_tcb.stk, taskstk;
addw end_os, start_os, elapsed;
addw elapsed, os_ovhd, os_ovhd;
move $10,
movl curr_tcb.pmt, pc;

ENDPROC;

; PROCEDURE sched_rtr_task (rtr_tcb : t_tcb)
; This procedure accepts a process control block and adds it to
; the ready to run queue in a prioritized fashion.
PROC sched_rtr_task;
popl taskstk, rtr_tcb;
movd $/ready/, rtr_tcb.status;
movw #/inhibit/, rtr_tcb.status;
addw rtr_tcb.sts.end, rtr_tcb.sts.begin, rtr_tcb.sts.begin;
addw rtr_tcb.sts.begin, rtr_tcb.sts.run_time, rtr_tcb.sts.run_time;
movw taskstk, rtr_tcb.sts.begin;

if (rtr_tcb.priority > curr_tcb.priority);
popl taskstk, rtr_tcb;
movl rtr_tcb.sts.end, rtr_tcb.sts.begin;
addw curr_tcb.sts.begin, curr_tcb.sts.run_time, curr_tcb.sts.run_time;
addw curr_tcb.sts.begin, curr_tcb.sts.run_time, curr_tcb.sts.run_time;
movw curr_tcb.sts.begin, curr_tcb.sts.end;

movd #/ready/, curr_tcb.status;
addl curr_tcb, rtr_tcb.priority;
call run_rtr_task;

ENDPROC;

; PROCEDURE get_sh (taskname : string)
; This procedure accepts a task name and sets the shared and
; sh_trc variables to hold the names of the task's messaging and trace
; structures.
PROC get_sh;
popl taskstk, tempstc;
movl $0, g_found;
lock;

if (g.size > 0) & (g_found);
addl g.size, g_st_elts;
if (g.st_elts == tempstc);
addl 1, g_found;
else;
addl nil_string, shared;
addl nil_string, shared;
elif;

ENDPROC;

; PROCEDURE get_local (taskname : string)
; This procedure accepts a task name and sets the local and
; loc_trc variables to hold the names of the task's messaging and trace
; structures.
PROC get_local;
pop taskstk, tempstr;
mov $0, g_found;
sizeq loc_comm_q, g_q_size;
while (g_q_size > $0) & ('g FOUND);
    delq loc_comm_q, g_q_size;
    if (g_st_elt_num = tempstr);
        mov $1, g_found;
    endif;
    addeq g_st_elt, loc_comm_q, $1;
    sub g_q_size, $1, g_q_size;
endwhile;

if (g_found = $1);
    mov g_st_elt.st_name, loc_str;
endif;

mov nil_string, local;
move nil_string, loc_str;

ENDPROC;

; PRIMITIVE nilprim

; This dummy primitive does nothing except pull in internode
; messages.

PRIM nilprim;
disable $10;
	
* time start_os;
* mov curr_tcb.sta.rdy_time, @rdy_str;
* mov curr_tcb.sta.wt_time, @wt_str;
* mov curr_tcb.sta.run_time, @run_str;

if (num_queued > $0);
    mov $0, done;
sizeq loc_comm_q, q_ax;
while (q_ax > $0) & ('done);
    delq loc_comm_q, st_elt;
    mov st_elt.st_name, local;

call get_sh, st_elt_num;
lock;
while (g_shared.num_msgs > $0);
    sub num_queued, $1, num_queued;
    sub g_shared.num_msgs, $1, g_shared.num_msgs;
    addq msg_elt, @local.msg_q, $1;
    if (glocal.waiting = $1);
        mov $0, @local.waiting;
move $1, done;
endif;
endwhile;
unlock;
addq st_elt, loc_comm_q, $1;
sub q_ax, $1, q_ax;
endwhile;

if (done = $1);
    * time start_os;
call mohed_trt_task, @local.block;
endif;
endif;
enable $10;

ENDPRIM;

; PRIMITIVE send (task_name : string; msg : string)

; This primitive accepts the name of a destination task and a
; message pointer. It then performs a nonblocking send operation.

PRIM send;
disable $10;
add $1, sends, sends;
*add $1, nn_send, nn_send;
* time start_os;
pop taskstk, msg_str;
pop taskstk, dest_str;
call get_local, dest_str;
if (local -> nil_string);
    if (glocal.waiting = $1);
        addq @msg_str, @local.msg_q, $1;

355
move $0, @local.waiting;
call sched_ptr_task, @local.block;
else:
  addq @msg_str, @local.msg_q, $1;
endif;
else:
call get_sh, dest_str;
lock;
add $1, num_queued, numqueued;
addq @msg_str, @shared.mgs, $1;
add $1, @shared.num_mgs, @shared.num_mgs;
unlock;
endif;
+time end_os;
+sub end_os, start_os, elapsed;
++add elapsed, tt_send, tt_send;
++add elapsed, os_ovhd, os_ovhd;
enable $10;
ENDPRIM;

; PRIMITIVE recv (msg_type : integer; @msg : string)

; This primitive performs a blocking nonselective receive operation.
; It accepts any type of message expected ($0 means accept any type)
; and a pointer to where the message is to be placed.
; The calling task is blocked
; until any message becomes available.

PASC block_recv;
+time curr_tcb.sts.finish;
+sub curr_tcb.sts.finish, curr_tcb.sts.begin, curr_tcb.sts.begin;
+add curr_tcb.sts.begin, curr_tcb.sts.run_time, curr_tcb.sts.run_time;
+move curr_tcb.sts.run_time, @run_str;
+time curr_tcb.sts.begin;
+time end_os;
+sub end_os, start_os, elapsed;
++add elapsed, tt_recv, tt_recv;
move $1, @local.waiting;
move @msg_str, curr_tcb.alt_str;
move $/wait, curr_tcb.status;
move taskstk, curr_tcb.stk;
add $2, pc, curr_tcb.pcnt;
move curr_tcb, @local.block;
call run_ptr_task;
disable $10;
move curr_tcb.alt_str, msg_str;
call get_local, currtask;
move $0, @local.waiting;
sizeq @local.msg_q, done;
if (done > $0)
delq @local.msg_q, @msg_str;
endif;
ENDPASC;

PRIM recv;
disable $10;
+time start_os;
+add $1, nn_recv, nn_recv;
pop taskstk, msg_str;
pop taskstk, msg_type;
call get_local, currtask;
call get_sh, currtask;
while (@shared.num_mgs > $0);
lock;
sub num_queued, $1, num_queued;
sub @shared.num_mgs, $1, @shared.num_mgs;
delq @shared.mgs, msg.alt;
delq msg.alt, @local.msg_q, $1;
unlock;
endwhile;
sizeq @local.msg_q, q.size;
if (q.size > $0:
call block_recv;
else;
delq @local.msg_q, @msg_str;

356
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, tt_recv, tt_recv;
*add elapsed, os_ovhd, os_ovhd;
endif;
enable #10;
ENDPRIM;

; PRIMITIVE sleep

; This primitive simulates the calling task by putting it in
; a sleep wait queue.
PRIM sleep;
    disable #10; /* time start_os;
pop taskstk, ret_pc;
    *time curr_tcb sts.finish;
*sub curr_tcb sts.finish, curr_tcb sts.begin, curr_tcb sts.begin;
*add curr_tcb sts.begin, curr_tcb sts.run_time, curr_tcb sts.run_time;
*move curr_tcb sts.run_time, @run_str;
*time curr_tcb sts.begin;
move #/wait/ curr_tcb status;
move taskstk, curr_tcb stk;
move ret_pc, curr_tcb.pcnt;
addc curr_tcb, await, curr_tcb.priority;
call run_str task;
ENDPRIM;

; PRIMITIVE get_data (task_name : string; data_name : string; size : integer)

; This primitive simulates data access between tasks. If the
; specified source task is not resident on the processor, then a number
; of shared accesses are made. One access is made for every 100 bytes
; of data requested.
PRIM get_data;
    disable #10; /* time start_os;
pop taskstk, data_size;
pop taskstk, msg_str;
pop taskstk, dest_str;
call get_local, dest_str;
if (local != nil_string);
    while (data_size > 0);
lock;
move @msg_str, q_size;
unlock;
sub data_size, #100, data_size;
endwhile;
else;
    while (data_size > 0);
move q_size, q_size;
sub data_size, #100, data_size;
endwhile;
endif;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_ovhd, os_ovhd;
enable #10;
ENDPRIM;

; PRIMITIVE end_resp

; This primitive simply calls response update to print the
; response traces.
PRIM endresp;
    disable #10;
pop taskstk, m_num;
*type #/response completed. response type : /
*type m_num;
*type #/ time : /
*type m_num;
*typecr m_num;
enable #10;
ENDPRIM;

; PRIMITIVE new_resp
This primitive clears the calling task’s current trace queue.

PRIM nearest;
ENDPRIM;

; PRIMITIVE evt_synch
; This primitive serves as the interrupt handler for the synchronization
; primitive. This interrupt is raised periodically every 8 seconds.

PRIM evt_synch;
disable $10;
ackint $0;
*add $1, evt_0, evt_0;
*time start_os;
sizeq swait, wt_chk;
move $0, found;
while ((wt_chk > $0) & (~found));
delq swait, temp_tcb;
if (temp_tcb.name = #/synch/);
move $1, found;
else;
addq temp_tcb, swait, temp_tcb.priority;
sub wt_chk, $1, wt_chk;
endif;
endwhile;
if (found);
**type #/synch change (0) at time : /;
**typecr start_os;
call sched_trx_task, temp_tcb;
endif;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_ovhd, os_ovhd;
enable $10;
ENDPRIM;

; PRIMITIVE evt_loss_synch
; This primitive serves as the interrupt handler for the synchronization
; loss primitive. This interrupt is raised sporadically to signal a loss of
; synchronization.

PRIM evt_loss_synch;
disable $10;
ackint $1;
*add $1, evt_1, evt_1;
*time start_os;
move $1, synch_loss;
sizeq swait, wt_chk;
move $0, found;
while ((wt_chk > $0) & (~found));
delq swait, temp_tcb;
if (temp_tcb.name = #/synch/);
move $1, found;
else;
addq temp_tcb, swait, temp_tcb.priority;
sub wt_chk, $1, wt_chk;
endif;
endwhile;
if (found);
**type #/synch loss (1) at time : /;
**typecr start_os;
call sched_trx_task, temp_tcb;
endif;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_ovhd, os_ovhd;
enable $10;
ENDPRIM;

; PRIMITIVE evt_msg_rx
; This primitive serves as the interrupt handler for the incoming external
; message interrupts. This interrupt is raised sporadically to signal the
; arrival of a message. It moves the MSG task ready to run from the
; blocked receive state.

PRIM evt_msg_rx;
disable #10;
ackint #2;
+add $1, evt_2, evt_3;
+time start_os;
mov $1, msg_ready;
rand $10, wt_chk;
if (wt_chk < #8);
    mov $1, msgtype;
else;
    if (wt_chk < #8);
        mov $2, msgtype;
    else;
        mov $3, msgtype;
endif;
endif;
mov #0, done;
call get_local, $/msg/;
if ($local.waiting = #1);
mov #0, $local.waiting;
mov $1, done;
endif;
if (done = #1);
+type $/msg event (2) of type : /;
+type msgtype;
+type $/ at time : /;
+typecr start_os;
call sched_rxt_task, $local.block;
endif;
+time end_os;
+sub end_os, start_os, elapsed;
+add elapsed, os_evhd, os_evhd;
enable $10;
ENDPRIM;

; PRIMITIVE evt_signal
;
This primitive serves as the interrupt handler for the signalling
interrupt handler. This interrupt is raised sporadically to signal the
arrival of a signalling event. It makes the SIGNAL task ready to run
from the blocked receive state.

PRIM evt_signal;
disable #10;
ackint #2;
+add $1, evt_2, evt_3;
+time start_os;
mov $1, sig_ready;
rand $10, wt_chk;
if (wt_chk < #8);
mov $6, sigtype;
else;
    if (wt_chk < #8);
mov $4, sigtype;
else;
mov $6, sigtype;
endif;
endif;
mov #0, done;
call get_local, $/signal/;
if ($local.waiting = #1);
mov #0, $local.waiting;
mov $1, done;
endif;
if (done = #1);
+type $/signal event (3) of type : /;
+type sigtype;
+type $/ at time : /;
+typecr start_os;
call sched_rxt_task, $local.block;
endif;
+time end_os;
+sub end_os, start_os, elapsed;
+add elapsed, os_evhd, os_evhd;
enable $10;
ENDPRIM;
; PRIMITIVE evt_clk
;
; This primitive serves as the interrupt handler for the clock interrupt
; used as a timing base for the audits task. This is a periodic interrupt.

PRIM evt_clk;
  disable #10;
  ackint #0;
  addq $1, evt_num, evt_num;
  time_start_os;
  addq swait, temp_tcb;
  move #0, found;
  while ((temp_tcb > #0) & (~found));
  delq swait, temp_tcb;
  if (temp_tcb.name = #/audits/);
    move #1, found;
  else;
    addq temp_tcb, swait, temp_tcb.priority;
    sub $1, temp_tcb, wt chk;
  endif;
endwhile;
if (found);
  *type #/audit event (4) at time : /;
  *typeerr start_os;
  call sched_trt_task, temp_tcb;
  endif;
  *time end_os;
  *sub end_os, start_os, elapsed;
  *add elapsed, os_ovhd, os_ovhd;
enable #10;
ENDPRIM;

BEGIN
  move #/node1/, mod_name;
  disable #10;
  move #/1_t1/, st elt.st_name;
  move #/l_t1.traces/, st elt.tloc_name;
  addq st elt, loc comm_q, #1;
  move #/l_t2/, st elt.st_name;
  move #/l_t2.traces/, st elt.tloc_name;
  addq st elt, loc comm_q, #1;
  move #/l_t3/, st elt.st_name;
  move #/l_t3.traces/, st elt.tloc_name;
  addq st elt, loc comm_q, #1;
  move #/l_t4/, st elt.st_name;
  move #/l_t4.traces/, st elt.tloc_name;
  addq st elt, loc comm_q, #1;
  move #/l_t5/, st elt.st_name;
  move #/l_t5.traces/, st elt.tloc_name;
  addq st elt, loc comm_q, #1;
  move #/l_t6/, st elt.st_name;
  move #/l_t6.traces/, st elt.tloc_name;
  addq st elt, loc comm_q, #1;
  move #/l_t7/, st elt.st_name;
  move #/l_t7.traces/, st elt.tloc_name;
  addq st elt, loc comm_q, #1;
  move #/l_t8/, st elt.st_name;
  move #/l_t8.traces/, st elt.tloc_name;
  addq st elt, loc comm_q, #1;
  move #/l_t9/, st elt.st_name;
  move #/l_t9.traces/, st elt.tloc_name;
  addq st elt, loc comm_q, #1;
  move #/l_t10/, st elt.st_name;
  move #/l_t10.traces/, st elt.tloc_name;
  addq st elt, loc comm_q, #1;
  move #/l_t11/, st elt.st_name;
  move #/l_t11.traces/, st elt.tloc_name;
  addq st elt, loc comm_q, #1;
  move #/l_t12/, st elt.st_name;
  move #/l_t12.traces/, st elt.tloc_name;
  addq st elt, loc comm_q, #1;
  move #/l_t13/, st elt.st_name;
  move #/l_t13.traces/, st elt.tloc_name;
  addq st elt, loc comm_q, #1;
  move #/l_t14/, st elt.st_name;
  move #/l_t14.traces/, st elt.tloc_name;
  addq st elt, loc comm_q, #1;

360
move #/t15/, st_elt.st_name;
mov #/t15.traces/, st_elt.trc_name;
addq st_elt, loc Comm q, #1;
mov #/t15/, st_elt.st_name;
mov #/t15.traces/, st_elt.trc_name;
addq st_elt, loc Comm q, #1;

if (nod_name = #/node1/)
lock;
move #/sh_st/, st_elt.st_name;
mov #/sh1.trc/, st_elt.trc_name;
addq st_elt, Comm q, #1;
mov #/sh12/, st_elt.st_name;
mov #/sh12.trc/, st_elt.trc_name;
addq st_elt, Comm q, #1;
mov #/sh13/, st_elt.st_name;
mov #/sh13.trc/, st_elt.trc_name;
addq st_elt, Comm q, #1;
mov #/sh14/, st_elt.st_name;
mov #/sh14.trc/, st_elt.trc_name;
addq st_elt, Comm q, #1;
mov #/sh15/, st_elt.st_name;
mov #/sh15.trc/, st_elt.trc_name;
addq st_elt, Comm q, #1;
mov #/sh16/, st_elt.st_name;
mov #/sh16.trc/, st_elt.trc_name;
addq st_elt, Comm q, #1;
mov #/sh17/, st_elt.st_name;
mov #/sh17.trc/, st_elt.trc_name;
addq st_elt, Comm q, #1;
mov #/sh18/, st_elt.st_name;
mov #/sh18.trc/, st_elt.trc_name;
addq st_elt, Comm q, #1;
mov #/sh19/, st_elt.st_name;
mov #/sh19.trc/, st_elt.trc_name;
addq st_elt, Comm q, #1;
mov #/sh10/, st_elt.st_name;
mov #/sh10.trc/, st_elt.trc_name;
addq st_elt, Comm q, #1;
mov #/sh11/, st_elt.st_name;
mov #/sh11.trc/, st_elt.trc_name;
addq st_elt, Comm q, #1;
mov #/sh12/, st_elt.st_name;
mov #/sh12.trc/, st_elt.trc_name;
addq st_elt, Comm q, #1;
mov #/sh13/, st_elt.st_name;
mov #/sh13.trc/, st_elt.trc_name;
addq st_elt, Comm q, #1;
mov #/sh14/, st_elt.st_name;
mov #/sh14.trc/, st_elt.trc_name;
addq st_elt, Comm q, #1;
mov #/sh15/, st_elt.st_name;
mov #/sh15.trc/, st_elt.trc_name;
addq st_elt, Comm q, #1;
mov #/sh16/, st_elt.st_name;
mov #/sh16.trc/, st_elt.trc_name;
addq st_elt, Comm q, #1;
mov #/sh17/, st_elt.st_name;
mov #/sh17.trc/, st_elt.trc_name;
addq st_elt, Comm q, #1;
mov #/sh18/, st_elt.st_name;
mov #/sh18.trc/, st_elt.trc_name;
addq st_elt, Comm q, #1;
mov #/sh19/, st_elt.st_name;
mov #/sh19.trc/, st_elt.trc_name;
addq st_elt, Comm q, #1;

unlock;
endif;

move tasklist, proclist;
siex proclist, br chk ;
mov curtask, tempsr;
move $1, wt chk;

while br chk <> $0;
delo proclist, tcr_name;
cxt w all name;
time curr_tcb.ats.begin;
mov wt chk, curr_tcb.num;
add $1, wt chk, wt chk;

if (tcb_name <> #/idle/)
lock;
*type nod_name;
delq loc Comm q, st elt;
mov tcb_name, st elt;
*type # --LOCAL-- TASK: /
*type tcb_name;

361
\*type \#/ QUEUE; /;
*type cr st_elx.st_name;
addq st_elx, temp_q, #1;
unlock;

lock:
    delq comm_q, st_elx;
    while (st_elx.nam > nil_string);
    addq st_elx, comm_q, #1;
    delq comm_q, st_elx;
endwhile;
move tcb_name, st_elx.nam;
*type \#/GLOBAL-- TASK: /;
*type tcb_name;
*type \#/ QUEUE; /;
*typecr st_elx.st_name;
addq st_elx, comm_q, #1;
unlock;
endif;
move taskstart, curr_tcb.pcnt;
move tcb_name, curr_tcb.name;
move \#/ready/, curr_tcb.status;
move currpuilo, curr_tcb.priority;
move taskstk, curr_tcb.stk;
addq curr_tcb, xtr, currprio;
sisq procclist, br_chk;
endwhile;
move temp_q, loc_comm_q;
move nil_queue, temp_q;

crtaw tempstx;
move \#/rdy_tm/, rdy_str;
move \#/wt_tm/ wt_str;
move \#/run_tm/ run_str;
*move \#/END RESPONSE................./, end_str;
bind evt_synch, #0;
bind evt_loss_synch, #1;
bind evt_mag_rx, #2;
bind evt_signal, #3;
bind evt_clk, #4;

enable #10;
call run_xtr_task;
ENDTASK;
ENDREXXE.
B.3.2 Application Software General Receive Kernel

In the discussed example, only the DIAGS, CINIT, and CHNLs tasks made use of the receive specific feature. To function without the receive specific feature, these tasks needed to be modified. The updated tasks are presented below.

```
; DIAGS Task

; STAT rdty tm, integer;
STAT st tm, integer;
STAT run tm, integer;
VAR msg out, t msg elt;
VAR msg in, t msg elt;
VAR waiting q, queue;
VAR msg pending, integer;
VAR q size, integer;
VAR dest, string;
VAR dg num, integer;
VAR tak start, integer;
VAR tak end, integer;
STAT cycle, integer;

ACTIVITY init;
   @time tak start;
ENDACT;

ACTIVITY upd_cycle;
   @time tak end;
   @sub tak end, tak start, cycle;
   @time tak start;
ENDACT;

ACTIVITY get msg;
   delq waiting q, msg in;
   misq waiting q, q size;
   if (q size = 0);
      move 0, msg pending;
   endif;
ENDACT;

ACTIVITY queue msg;
   addq msg in, waiting q, 1;
   move 1, msg pending;
ENDACT;

ACTIVITY chnl req;
   move $12, msg out type;
   move $/diags, msg out arc;
ENDACT;

ACTIVITY choose diag;
   rend $4, dg num;
   move $21, msg out type;
   move $/diags, msg out arc;
   if (dg num = 0);
      move $/bondiag, dest;
   else;
      if (dg num = 1);
         move $/chndiag, dest;
      else;
         if (dg num = 2);
            move $/msgdiag, dest;
         else;
            move $/tidiag, dest;
         endif;
      endif;
   endif;
ENDACT;

BEGIN
```

363
SETSTATE #/clear/;
ENDIF;
ENDSTATE;
ENDSELECT;
ENDLOOP
ENDTASK;

; CINIT Task
STAT rdY_tm, integer;
STAT ut_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_elt;
VAR msg_in, t_msg_elt;
VAR waiting_q, queue;
VAR msg_pending, integer;
VAR q_size, integer;
VAR task_start, integer;
VAR task_end, integer;
STAT cycle, integer;

ACTIVITY init;
*time task_start;
move 00, msg_pending;
ENDACT;

ACTIVITY upd_cycle;
*time task_end;
*sub task_end, task_start, cycle;
*time task_start;
ENDACT;

ACTIVITY get_mag;
delq waiting_q, msg_in;
sizeq waiting_q, q_size;
if (q_size = 00);
mov 00, msg_pending;
edif;
ENDACT;

ACTIVITY queue_mag;
addq msg_in, waiting_q, #1;
mov #1, msg_pending;
ENDACT;

ACTIVITY conn_req;
mov #1, msg_out.type;
mov #/init, msg_out.arc;
ENDACT;

ACTIVITY chnl_req;
mov #12, msg_out.type;
mov #/init, msg_out.arc;
ENDACT;

ACTIVITY stat_req;
mov #16, msg_out.type;
mov #/init, msg_out.arc;
ENDACT;

BEGIN
DO.ACT init;
callp nilprim;
SETSTATE #/clear/;
LOOP:
STATESELECT;

STATE clear;
if #true;
DO.ACT upd_cycle;
callp nextrep;
SETSTATE #/strt/;
edif;
ENDSTATE;

STATE chnl;
if #true;
callp send, #/chnl/;
SETSTATE #/acquire/;
edif;
ENDSTATE;

STATE get_mag;
if #true;
callp init_msg;
callp nilprim;
SETSTATE #/msg_clr/;
edif;
ENDSTATE;

STATE ut_ack;
if #true;
callp recv, #0, #msg_in/;
SETSTATE #/msg_clr/;
edif;
ENDSTATE;

STATE chnl_ack;
if (msg_in.type = #14);
callp nilprim;
SETSTATE #/chnl/;
edif;
ENDSTATE;

STATE INIT;
if #true;
callp connect, #/msg_out/;
SETSTATE #/init/;
edif;
ENDSTATE;

STATE msg_clr;
if #true;
callp send, #/msg_clr/;
SETSTATE #/clear/;
edif;
ENDSTATE;

STATE msg_init;
if #true;
callp stat_msg;
callp nilprim;
SETSTATE #/msg_init/;
edif;
ENDSTATE;

STATE msg_init;
if #true;
callp msg_init;
SETSTATE #/msg_init/;
edif;
ENDSTATE;

STATE clear;
if #true;
callp upd_cycle;
callp nextrep;
SETSTATE #/strt/;
edif;
ENDSTATE;

STATE chnl;
if #true;
callp send, #/chnl/;
SETSTATE #/chnl/;
edif;
ENDSTATE;

STATE init;
if #true;
callp init_msg;
callp nilprim;
SETSTATE #/init/;
edif;
ENDSTATE;

STATE clear;
if #true;
callp upd_cycle;
callp nextrep;
SETSTATE #/strt/;
edif;
ENDSTATE;

STATE chnl;
if #true;
callp send, #/chnl/;
SETSTATE #/acquire/;
edif;
ENDSTATE;

STATE msg_clr;
if #true;
callp send, #/msg_clr/;
SETSTATE #/clear/;
edif;
ENDSTATE;

STATE msg_init;
if #true;
callp send, #/msg_init/;
SETSTATE #/msg_init/;
edif;
ENDSTATE;

STATE clear;
if #true;
callp upd_cycle;
callp nextrep;
SETSTATE #/strt/;
edif;
ENDSTATE;

STATE chnl;
if #true;
callp send, #/chnl/;
SETSTATE #/chnl/;
edif;
ENDSTATE;

STATE msg_clr;
if #true;
callp send, #/msg_clr/;
SETSTATE #/clear/;
edif;
ENDSTATE;

STATE msg_init;
if #true;
callp send, #/msg_init/;
SETSTATE #/msg_init/;
edif;
ENDSTATE;

STATE clear;
if #true;
callp upd_cycle;
callp nextrep;
SETSTATE #/strt/;
edif;
ENDSTATE;
STATE chk_ack;
IF (send_ack = $1);
DO.ACT clear_ack;
CALLP nilprim;
SETSTATE #/clear/;
ENDIF;
ENDACT;
ENDSTATE;
ENDSELECT;
ENDLOOP
ENDTASK;

STATE clear;
IF #true;
DO.ACT upd_cycle;
CALLP nevresp;
SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
IF #true;
CALLP recv, $0, $/msg_in/;
SETSTATE #/got_msg/;
ENDIF;
ENDSTATE;

STATE get_msg;
IF (msg_in.type = $12);
DO.ACT get_chnl;
CALLP nilprim;
SETSTATE #/chk_ack/;
ENDIF;
IF #true;
DO.ACT free_chnl;
CALLP nilprim;
SETSTATE #/chk_ack/;
ENDIF;
ENDSTATE;
B.3.3 Blocking Send and Receive Kernel

KERNEL CODE FILE

This kernel contains the following primitives:
blocking send
blocking receive
sleep
get data from another task
nilprim

The following external message types are supported:

@ - start call setup
@2 - provide tone
@3 - run diagnostic
@4 - integrity ok
@5 - integrity failure
@6 - call takedown

The following internal message types are supported:

@1 - START - request call setup
@2 - TONES - request for tone
@3 - DIAG - request single random diagnostic
@4 -
@5 -
@6 -
@7 - MSG - report fault
@8 - MSG - perform and report sync operation
@9 - FAULT - synch loss
@10 - SIGNAL - general fault report
@11 - CONN - perform connect/disconnect
@12 - CANS - allocate channel
@13 - CANS - free channel
@14 - all_tasks - channel granted
@15 - STAT - call setup complete
@16 - SIGNAL - start integrity checking
@17 - STAT - integrity failure report
@18 - END - caller hung up
@19 - STAT - call finished
@20 - STAT - stop integrity checking
@21 - all_diags - run diagnostic
@22 - provide tone - stop carrier checking
@23 - TINTC - monitor carriers

The following response types are supported:

@1 - MSG - fault report sent to switching center
@2 - MSG - sync source changed
@3 - CONN - switching center tone request connected
@4 - STAT - call finished
@5 - STAT - call started
@6 - all_diags - diagnostic completed without failure
@7 - TINTC - carrier scan completed without failure

KERNEL baseb;

TASK kern_task, SQ;

TYPE t_tcb_stat = rdy_time : integer *;
run_time : integer *;
uc_time : integer *
beg : integer *;
finish : integer *;
resp_q : queue;

TYPE t_shared_state = data_trace :queue;

TYPE t_tcb = name : string *;
num : integer *;
status : string *;
priority : integer *;
pont : integer *;
elt : string *
stk : stack *;
sta : t_tcb_status;

TYPE t_msg_wt = msg : string *;
end : string;

TYPE t_recv_stat = r_wait : boolean *;
r_msg : string *;
r_wid : string;

366
TYPE t_mag_struct = task_num : string *;
  msgs : queue *;
  rcv_stat : t_mag_struct *;
  reply : string *;
  tcb_ptr : t_tcb;
msg queue holding area
msg receive status
ptr to reply area
associated tcb wait area

TYPE t_sh_data = task_num : string *;
task name
num_msg : integer *;
msgs : queue *;
msg queue (t_mag_elm)
trc : queue *;	race queue (t_trc_elm)

TYPE t_id_struct = num : string *;
task name : string *
sat_name : string *

SHARED comm_q, queue; of t_id_struct (one elt per task)
SHARED sh_id, t_sh_data;
SHARED sh_id2, t_sh_data;
SHARED sh_id3, t_sh_data;
SHARED sh_id4, t_sh_data;
SHARED sh_id5, t_sh_data;
SHARED sh_id6, t_sh_data;
SHARED sh_id7, t_sh_data;
SHARED sh_id8, t_sh_data;
SHARED sh_id9, t_sh_data;
SHARED sh_id10, t_sh_data;
SHARED sh_id11, t_sh_data;
SHARED sh_id12, t_sh_data;
SHARED sh_id13, t_sh_data;
SHARED sh_id14, t_sh_data;
SHARED sh_id15, t_sh_data;
SHARED sh_id16, t_sh_data;
SHARED sh_id17, t_sh_data;
SHARED sh_id18, t_sh_data;
SHARED stat_data, integer;
SHARED num_queue, integer;

; Message queues
VAR loc_comm_q, queue; of t_id struct (one elt per local task)
VAR st_alt, t_id_struct;
VAR l_51, t_mag_struct;
VAR l_52, t_mag_struct;
VAR l_53, t_mag_struct;
VAR l_54, t_mag_struct;
VAR l_55, t_mag_struct;
VAR l_56, t_mag_struct;
VAR l_57, t_mag_struct;
VAR l_58, t_mag_struct;
VAR l_59, t_mag_struct;
VAR l_60, t_mag_struct;
VAR l_61, t_mag_struct;
VAR l_62, t_mag_struct;
VAR l_63, t_mag_struct;
VAR l_64, t_mag_struct;
VAR l_65, t_mag_struct;
VAR l_66, t_mag_struct;
VAR l_67, t_mag_struct;
VAR l_68, t_mag_struct;
VAR l_69, t_mag_struct;
VAR l_70, t_mag_struct;
VAR l_71, t_mag_struct;
VAR l_72, t_mag_struct;
VAR l_73, t_mag_struct;
VAR l_74, t_mag_struct;
VAR l_75, t_mag_struct;
VAR l_76, t_mag_struct;
VAR l_77, t_mag_struct;
VAR l_78, t_mag_struct;
VAR l_79, t_mag_struct;
VAR l_80, t_mag_struct;
VAR l_81, t_mag_struct;
VAR l_82, t_mag_struct;
VAR l_83, t_mag_struct;
VAR l_84, t_mag_struct;
VAR l_85, t_mag_struct;
VAR l_86, t_mag_struct;
VAR l_87, t_mag_struct;
VAR l_88, t_mag_struct;
VAR l_89, t_mag_struct;
VAR l_90, t_mag_struct;
VAR l_91, t_mag_struct;
VAR l_92, t_mag_struct;
VAR l_93, t_mag_struct;
VAR l_94, t_mag_struct;
VAR l_95, t_mag_struct;
VAR l_96, t_mag_struct;
VAR l_97, t_mag_struct;
VAR l_98, t_mag_struct;
VAR l_99, t_mag_struct;
VAR l_100, t_mag_struct;
VAR l_101, t_mag_struct;
VAR l_102, t_mag_struct;
VAR l_103, t_mag_struct;
VAR l_104, t_mag_struct;
VAR l_105, t_mag_struct;
VAR local, string;
VAR shared, string;
VAR loc_trc, string;
VAR sh_trc, string;

STAT sends, integer;
STAT os_ordh, integer;
STAT switches, integer;
STAT evt_0, integer;
STAT evt_1, integer;
STAT evt_2, integer;
STAT evt_3, integer;
STAT evt_4, integer;
VAR start_os, integer;
VAR end_os, integer;
VAR elapse, integer;
VAR rdy_str, string;
VAR wt_str, string;
VAR run_str, string;
VAR end_str, string;
VAR wait, queue;
VAR chk_bool, boolean;
VAR source, string;
VAR rep_str, string;
VAR msg_wait, t_msg_wait;
VAR ut_ch, integer;
VAR msg_str, string;
VAR dest_str, string;
VAR temp_tcb, t_tcb;
VAR temp2_tcb, t_tcb;
VAR ret_pc, integer;
VAR tmp_stk, stack;
VAR found, integer;
VAR data_ssize, integer;
VAR nil_queue, queue;
VAR nil_string, string;
VAR msg_int, t_msg_int;
VAR m_num, integer;
VAR mtype, integer;
VAR q_len, integer;
VAR done, integer;

; Global Kernel Variables
VAR curr_tcb, t_tcb ; current task control block
VAR proclist, queue ; copy of task name list
VAR br_chk, integer ; branch condition check
VAR temp_tcb, string ; temporary string value
VAR tcb_name, string ; current task name
VAR rtr, queue ; ready to run queue

; Variables for get_local and get_sh procedures
VAR g_q_size, integer;
VAR g_found, integer;
VAR g_st_elts, t_id_struct;
VAR q_size, integer ; size of trace queue
VAR temp_q, queue ; data response queue copy

; VARIABLE declarations for procedure RUN_RTR_TASK
VAR error, boolean; Set if no ready processes

; VARIABLE declarations for procedure SCHED_RTR_TASK
VAR rtr_pcb, t_tcb; PCB to be scheduled

; PROCEDURE run_rtr_task
; This procedure takes the first ready to run task off the rtr
; queue (this is the one with the highest priority) and copies
; its saved state into the current processor state. If there are
; no ready to run tasks, then an error message is printed and
; the kernel terminates.
PROC run_rtr_task;
   sizeq rtr, br_chk;
   IF (br_chk = 0);
      printer #/ERROR: NO RTR PROCESSES/;
      HALT;
   ELSE
      DELQ rtr, curr_tcb;
      move curr_tcb.name, tcb_name;
      extr tcb_name;
      +add #1, switches, switches;
      +time curr_tcb.sta.finish, curr_tcb.sta.begin, curr_tcb.sta.begin;
      +add curr_tcb.sta.begin, curr_tcb.sta.rd_time, curr_tcb.sta.rd_time;
      +move curr_tcb.sta.rd_time, &rdy_str;
      +move curr_tcb.sta.ut_time, &ut_str;
      +time curr_tcb.sta.begin;
      move #/running/, curr_tcb.status;
      move curr_tcb.stk, taskstk;
      +time end_os;
      +sub end_os, start_os, elapsed;
      +add elapsed, os_ovhd, os_ovhd;
      enable 510;
      move curr_tcb.pcnt, pc;
   ENDIF;
ENDPROC;
PROCEDURE sched_rtr_task (rtr_pcb : t_pcb)

This procedure accepts a process control block and adds it to
the ready to run queue in a prioritized fashion.

PROC sched_rtr_task;
  pop  taskstk,  rtr_pcb;
  move $/ready/,  rtr_pcb.latgest;
  *time  rtr_pcb.latgest.finish;
  *sub  rtr_pcb.latgest.finish, rtr_pcb.latgest.begin, rtr_pcb.latgest.begin;
  *add  rtr_pcb.latgest.begin, rtr_pcb.latgest.st_time, rtr_pcb.latgest.st_time;
  *move  taskstk,  addr;
  *time  rtr_pcb.latgest.begin;
  *addq  rtr_pcb.latgest,  rtr_pcb.latgest.priority;
  if  (rtr_pcb.priority > curr_tcb.priority);
    pop  taskstk,  ret_pc;
    pop  taskstk,  ret_pc;
    move  ret_pc,  curr_tcb.sparse;
    move  taskstk,  curr_tcb.task;
    *time  curr_tcb.task.finish;
    *sub  curr_tcb.task.finish, curr_tcb.task.begin, curr_tcb.task.begin;
    *add  curr_tcb.task.begin, curr_tcb.task.run_time, curr_tcb.task.run_time;
    *move  curr_tcb.task.begin;
    *time  curr_tcb.task.begin;
    move $/ready/,  curr_tcb.status;
    *addq  curr_tcb.task,  curr_tcb.priority;
    call  run_rtr_task;
  halt;
endif;
ENDPROC;

PROCEDURE get_sh (taskname : string)

This procedure accepts a task name and sets the shared and
sh_tcb variables to hold the names of the task's messaging and trace
structures.

PROC get_sh;
  pop  taskstk,  tempstr;
  move  $9,  g_found;
  lock;
  sizeq  comm_q, g.q_size;
  while  (g.q_size > $9) & (*g_found);
    delq  comm_q, g.st_elt;
    if  (g.st_elt.lat_name = tempstr);
      move $1,  g_found;
    endif;
    *addq  g.st_elt,  comm_q, $1;
    *sub  g.q_size,  $1,  g.q_size;
    endwhile;
  unlock;
  if  (g_found = $1);
    move  g.st_elt.lat_name,  shared;
    move  g.st_elt.src_name,  sh_tcb;
  else;
    nil_string,  shared;
    move  nil_string,  sh_tcb;
  endif;
ENDPROC;

PROCEDURE get_local (taskname : string)

This procedure accepts a task name and sets the local and
loc_tcb variables to hold the names of the task's messaging and trace
structures.

PROC get_local;
  pop  taskstk,  tempstr;
  move  $9,  g_found;
  sizeq  loc_comm_q, g.q_size;
  while  (g.q_size > $9) & (*g_found);
    delq  loc_comm_q, g.st_elt;
    if  (g.st_elt.lat_name = tempstr);
      move $1,  g_found;
  ...
endif;
addq g_st_elt, loc_comm_q, $1;
sub g_q_size, $1, g_q_size;
endif;

if (g_ffound = $1);
move g_st_elt.st_name, local;
move g_st_elt.tcb_name, loc_tcb;
else;
move nil_string, local;
move nil_string, loc_tcb;\nendif;
ENDPROC;

: PRIMITIVE nilprim
: This dummy primitive does nothing except pull in intmode
: messages.

PRIM nilprim:
disable $10;
*time start_cs;
*move curr_tcb.sta.rdy_time, &rdy_str;
*move curr_tcb.sta.run_time, &run_str;
if (num_queued > 0);
move $0, done;
sizeq loc_comm_q, q_sz;
while (q_sz > 0) & (*done);
delq loc_comm_q, st_elt;
move st_elt.st_name, local;

call get_sh, st_elt.name;
lock;
if (*shared.num_msgs > 0);
move *sharedmsgs, temp_q;
delq temp_q, msg_elt;
move nil_queue, temp_q;
move Glocal.rcv_st.r_wait, ckt_boot;
if (msg_elt.rep = $1) | (ckt_boot);

sub num_queued, $1, num_queued;
sub *sharedmsgs, $1, *sharedmsgs;
delq *sharedmsgs, msg_elt;
move $1, done;
if (msg_elt.rep = $1);
move curr_task, tempstr;
cxsw st_elt.name;
move Glocal.reply, source;
move msg_elt, *source;
cxsw tempstr;
else;
move curr_task, tempstr;
cxsw st_elt.name;
move Glocal.rcv_st.r_msg, source;
move msg_elt, *source;
move Glocal.rcv_st.r_sid, source;
move msg_elt.src, *source;
cxsw tempstr;
move false, Glocal.rcv_st.r_wait;

endif;
endif;

unlock;
addq st_elt, loc_comm_q, $1;
sub q_sz, $1, q_sz;
endif;
endif;
enable $10;
ENDPRIM;

: PRIMITIVE send (task_name : string; Msg : string; Creply : string)
: This primitive accepts the name of a destination task and a
: message pointer. It then performs a blocking send operation.
PRIM send;
  disable $10;
  add $1, sends, sends;
  +time start_os;
  pop taskstk, rep_str;
  pop taskstk, msg_str;
  pop taskstk, dest_str;
  call get_local, dest_str;
  if (local <> nil_string);
    move Global.recv_str.r_wait, chk_bool;
    if (chk_bool);
      move $msg_str, msg_elt;
      move curr_task, tempstr;
      octau dest_str;
      move Global.recv_str.r_msg, source;
      move msg_elt, resource;
      move Global.recv_str.r_cid, source;
      move tempstr, Gsource;
      octau tempstr;
      move $false, Global.recv_str.r_wait;
      call sched_str_task, Global.tcb_ptr;
    else;
      move msg_str, msg_wait.msg_i;
      move curr_task, msg_wait.end_i;
      addq msg_wait, Global.mage, $1;
    endif;
  else;
    call get_sh, dest_str;
    lock;
    add $1, num_queued, num_queued;
    move $0, $msg_str.repl;
    addq $msg_str, $shared.mages, $1;
    add $1, $shared.num_mage,
    unlock;
  endif;
  call get_local, curr_task;
  move rep_str, Global.replse;
  pop taskstk, rep_pc;
  move $/wait/, curr_tcb.status;
  move rep_pc, curr_tcb.pcnt;
  move taskstk, curr_tcb.tkb;
  move curr_tcb, Global.tcb_ptr;
  +time end_os;
  +sub end_os, start_os, elapsed;
  add elapsed, os_ord, os Ord;
  +time curr_tcb.tcs.finish;
  +sub curr_tcb.tcs.finish, curr_tcb.tcs.begin, curr_tcb.tcs.begin;
  +add curr_tcb.tcs.begin, curr_tcb.tcs.run_time, curr_tcb.tcs.run_time;
  move curr_tcb.tcs.run_time, $run_str;
  +time curr_tcb.tcs.begin;
  call run_str_task;
ENDPRIM;

; PRIMIVE recv ($msg : string; $send_id : string)
;    This primitive performs a blocking receive operation.
;    It accepts the type of message expected ($0 means accept any type),
;    a pointer to where the message is to be placed, and a pointer to
;    where the sender's id is to be placed.
;    The calling task is blocked
;    until any message becomes available.
PRIM recv:
  disable $10;
  +time start_os;
  pop taskstk, dest_str;
  pop taskstk, msg_str;
  call get_local, curr_task;
  call get_sh, curr_task;
  if ($shared.num_mage > $0);
    lock;
    sub num_queued, $1, num_queued;
    sub $shared.num_mage, $1, $shared.num_mage;
    delq $shared.mages, msg_elt;
    unlock;
    move msg_elt, $msg_str;
    move msg_elt, ptr, dest_str;
    +time end_os;
  endif;

371
*sub end_os, start_os, elapsed;
*add elapsed, os_ohd, os_ohd;
enable $10;
ret;
else:
    elseq $localmsgs, q_size;
if (q_size > 0):
    deq $localmsgs, msg_wait;
    move current_task, tempstr;
    ctsw msg_wait, end_1;
    move msg_wait, msg_1, source;
    move source, msg_el;
    ctsw tempstr;
    move msg_el, $msg_str;
    move msg_wait, end_1, $dest_str;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_ohd, os_ohd;
enable $10;
ret;
else:
    move $true, $localrecvstr_wait;
    move $msg_str, $localrecvstr_msg;
    move dest_str, $localrecvstr_sid;
    pop taskstk, ret_pc;
    move $/wait, current_tcb.status;
    move ret_pc, current_tcb.pcnt;
    move taskstk, current_tcb.stk;
    move current_tcb, $local_tcb_ptr;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_ohd, os_ohd;
*time current_tcb.start;
*sub current_tcb.start, current_tcb.start.begin, current_tcb.start.begin;
*add current_tcb.start.begin, current_tcb.start.run_time, current_tcb.start.run_time;
*move current_tcb.start.run_time, @run_tcb;
*time current_tcb.start.begin;
call run_str_task;
endif;
endif;
ENDPRIM;

; PRIMITIVE reply (task_name : string; msg : string)
;
This primitive accepts the name of a destination task and a message pointer. It then performs a reply operation freeing the specified task.
PRIM reply;
disable $10;
*time start_os;
*add $1, sends, sends;
pop taskstk, msg_str;
pop taskstk, dest_str;
call get_local, dest_str;
if (local <> nil_string):
    move current_task, tempstr;
    move $msg_str, msg_el;
    ctsw dest_str;
    move $globalreply, source;
    move msg_el, @source;
    ctsw tempstr;
call sched_rtr_task, $local_tcb_ptr;
else:
call get_sh, dest_str;
lock;
*add $1, num_queued, num_queued;
move $1, $msg_str.rep;
add $msg_str, $sharedmsgs, $2;
add $1, $sharednummsgs, $sharednummsgs;
unlock;
endif;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_ohd, os_ohd;
enable $10;
ENDPRIM;

; PRIMITIVE sleep

372
This primitive simply blocks the calling task by putting it in the sleep wait queue.

PRIM sleep;
  disable $10;
  @time start_os;
  pop taskstk, ret_pc;
  @time curr_tcb sts.finish;
  sub curr_tcb sts.finish, curr_tcb sts.begin, curr_tcb sts.begin;
  add curr_tcb sts.begin, curr_tcb sts run time, curr_tcb sts run time;
  move curr_tcb sts run time, $run str;
  @time curr_tcb sts.begin;
  move $wait/, curr_tcb status;
  move taskstk, curr_tcb stk;
  move ret_pc, curr_tcb pont;
  addq curr_tcb, await, curr_tcb priority;
  call run str task;
ENDPRIM;

PRIMITIVE get data (task_name : string; data_name : string; size : integer)
This primitive simulates data access between tasks. If the specified source task is not resident on the processor, then a number of shared accesses are made. One access is made for every 100 bytes of data requested.

PRIM get data;
  disable $10;
  @time start os;
  pop taskstk, data size;
  pop taskstk, mag str;
  pop taskstk, dest str;
  call get_local, dest str;
  if (local = nil string);
    while (data size > $0);
      lock;
      move $mag str, q size;
      unlock;
      sub data size, $100, data size;
      endwhile;
    else;
      while (data size > $0);
        move q size, q size;
        sub data size, $100, data size;
        endwhile;
      endif;
  @time end os;
  sub end os, start os, elapsed;
  add elapsed, os_ovhd, os_ovhd;
  enable $10;
ENDPRIM;

PRIMITIVE end resp
This primitive simply calls response update to print the response traces.

PRIM end resp;
  disable $10;
  pop taskstk, m num;
  @type $/ response completed. Type : !;
  @type m num;
  @time m num;
  @type $/ Time : !;
  @typeperc m num;
  enable $10;
ENDPRIM;

PRIMITIVE new resp
This primitive clears the calling task's current trace queue.

PRIM new resp;
ENDPRIM;

PRIMITIVE ev_t sync
This primitive serves as the interrupt handler for the synchronization primitive. This interrupt is raised periodically every 5 seconds.

```plaintext
PRIM evt_synch:
    disable $10;
    ackint $0;
    *add $1, evt_0, evt_0;
    *time start_cs;
    sizeq wait, wt_chk;
    move $0, found;
    while ((wt_chk > $0) & (*found));
    delq wait, temp_tcb;
    if (temp_tcb.name = #/synch/);
        move $1, found;
    else;
        addq temp_tcb, wait, temp_tcb.priority;
        sub wt_chk, $1, wt_chk;
    endif;
    endwhile;
    if (*found);
        *type #/synch event (O) at time : /;
        *typecr start_cs;
        call sched_vtr_task, temp_tcb;
    endif;
    *time end_cs;
    *sub end_cs, start_cs, elapsed;
    *add elapsed, os_ovhd, os_ovhd;
    enable $10;
ENDPRIM;
```

This primitive serves as the interrupt handler for the synchronization loss primitive. This interrupt is raised sporadically to signal a loss of synchronization.

```plaintext
PRIM evt_loss_synch:
    disable $10;
    ackint $1;
    *add $1, evt_1, evt_1;
    *time start_cs;
    sizeq wait, wt_chk;
    move $0, found;
    while ((wt_chk > $0) & (*found));
    delq wait, temp_tcb;
    if (temp_tcb.name = #/synch/);
        move $1, found;
    else;
        addq temp_tcb, wait, temp_tcb.priority;
        sub wt_chk, $1, wt_chk;
    endif;
    endwhile;
    if (*found);
        *type #/synch loss (1) at time : /;
        *typecr start_cs;
        call sched_vtr_task, temp_tcb;
    endif;
    *time end_cs;
    *sub end_cs, start_cs, elapsed;
    *add elapsed, os_ovhd, os_ovhd;
    enable $10;
ENDPRIM;
```

This primitive serves as the interrupt handler for the incoming external message interrupt. This interrupt is raised sporadically to signal the arrival of a message. It makes the MSG task ready to run from the blocked receive state.

```plaintext
PRIM evt_msg_rx:
    disable $10;
    ackint $2;
    *add $1, evt_2, evt_2;
    *time start_cs;
    move $1, msg_cdy;
    rand $10, wt chk;
    if (wt chk < $6);
```
move $1, msgtype;
else:
  if (st_chk < $0):
    move $2, msgtype;
  else:
    move $3, msgtype;
  endif;
endif;
move $0, done;
call get_local, #/msg/;
move @local.recv_msg_wait, chk_bool;
if (chk_bool):
  move $false, @local.recv_msg_wait;
  move $1, done;
endif;
if (done = $1):
  *type #/msg; event (2) of type : /;
  *type msgtype;
  *type $/at time : /
  *typecr start_cs;
  call sched_rq_task, @local_tcb_ptr;
  endif;
  *time end_cs;
  *sub end_cs, start_cs, elapsed;
  *add elapsed, os_ovhd, os_ovhd;
  enable $10;
ENDPRI;

; PRIMITIVE evt_signal
;
; This primitive serves as the interrupt handler for the signalling
; interrupt handler. This interrupt is raised sporadically to signal the
; arrival of a signalling event. It makes the SIGNAL task ready to run
; from the blocked receive state.

PRIM evt_signal:
  disable $10;
  addint $3;
  *add $1, evt_3, evt_3;
  *time start_cs;
  move $1, sig_by;
  rand $10, wc.chk;
  if (wt_chk < $0):
    move $6, sigtype;
    else:
      if (wc_chk < $0):
        move $4, sigtype;
      else:
        move $5, sigtype;
      endif;
      endif;
move $0, done;
call get_local, #/signal/;
move @local.recv_msg_wait, chk_bool;
if (chk_bool):
  move $false, @local.recv_msg_wait;
  move $1, done;
endif;
if (done = $1):
  *type #/sig event (3) of type : /;
  *type sigtype;
  *type $/at time : /
  *typecr start_cs;
  call sched_rq_task, @local_tcb_ptr;
  endif;
  *time end_cs;
  *sub end_cs, start_cs, elapsed;
  *add elapsed, os_ovhd, os_ovhd;
  enable $10;
ENDPRI;

; PRIMITIVE evt_clk
;
; This primitive serves as the interrupt handler for the clock interrupt
; used as a timing base for the audit task. This is a periodic interrupt.

PRIM evt_clk:
  disable $10;
block #4;
  load #1, evb.4, evr.4;
  time start.os;
  saveq wait, wt_chk;
move #0, found;
while ((wt_chk > #0) & ("found"));
deq wait, temp_tcb;
if (temp_tcb.name = $/audits/);
movement found;
else:
  add temp_tcb, wait, temp_tcb.priority;
  sub wt_chk, #1, wt_chk;
endif;
endwhile;
if (found);
  #audit event (4) at time : /
  #type start_os;
call sched_trx_task, temp_tcb;
endif;
  #time end_os;
  sub end_os, start_os, elapsed;
  add elapsed, os_ovhd, os_ovhd;
end #10;
ENDPRIM;
BEGIN
move #/nodes1/, nod_name;
disable #10;
movew #/i.T1/, st elt.st_name;
movew #/i.T2/ traces/, st elt.trc_name;
addq st elt, loc.comm.q, #1;
movew #/i.T3/, st elt.st_name;
movew #/i.T4/ traces/, st elt.trc_name;
addq st elt, loc.comm.q, #1;
movew #/i.T5/, st elt.st_name;
movew #/i.T6/ traces/, st elt.trc_name;
addq st elt, loc.comm.q, #1;
movew #/i.T7/, st elt.st_name;
movew #/i.T8/ traces/, st elt.trc_name;
addq st elt, loc.comm.q, #1;
movew #/i.T9/, st elt.st_name;
movew #/i.T10/ traces/, st elt.trc_name;
addq st elt, loc.comm.q, #1;
movew #/i.T11/, st elt.st_name;
movew #/i.T12/ traces/, st elt.trc_name;
addq st elt, loc.comm.q, #1;
movew #/i.T13/, st elt.st_name;
movew #/i.T14/ traces/, st elt.trc_name;
addq st elt, loc.comm.q, #1;
movew #/i.T15/, st elt.st_name;
movew #/i.T16/ traces/, st elt.trc_name;
addq st elt, loc.comm.q, #1;
movew #/i.T17/, st elt.st_name;
movew #/i.T18/ traces/, st elt.trc_name;
addq st elt, loc.comm.q, #1;
movew #/i.T19/, st elt.st_name;
movew #/i.T20/ traces/, st elt.trc_name;
addq st elt, loc.comm.q, #1;
movew #/i.T21/, st elt.st_name;
movew #/i.T22/ traces/, st elt.trc_name;
addq st elt, loc.comm.q, #1;
movew #/i.T23/, st elt.st_name;
movew #/i.T24/ traces/, st elt.trc_name;
addq st elt, loc.comm.q, #1;
movew #/i.T25/, st elt.st_name;
movew #/i.T26/ traces/, st elt.trc_name;
addq st elt, loc.comm.q, #1;
movew #/i.T27/, st elt.st_name;
movew #/i.T28/ traces/, st elt.trc_name;
addq st elt, loc.comm.q, #1;
movew #/i.T29/, st elt.st_name;
movew #/i.T30/ traces/, st elt.trc_name;
addq st elt, loc.comm.q, #1;
movew #/i.T31/, st elt.st_name;
movew #/i.T32/ traces/, st elt.trc_name;
addq st elt, loc.comm.q, #1;
movew #/i.T33/, st elt.st_name;
movew #/i.T34/ traces/, st elt.trc_name;
addq st elt, loc.comm.q, #1;
movew #/i.T35/, st elt.st_name;
movew #/i.T36/ traces/, st elt.trc_name;
addq st elt, loc.comm.q, #1;
if (nod_name = #/nodes1/);
lock;
move #sh_t1/, st_elt.st_name;
move #sh_t2, st_elt.st_name;
addq st_elt, comm_q, $1;
mv #sh_t3/, st_elt.st_name;
move #sh_t4, st_elt.trc/.
addq st_elt, comm_q, $1;
mv #sh_t5/, st_elt.st_name;
move #sh_t5.trc/.
addq st_elt, comm_q, $1;
mv #sh_t6/, st_elt.st_name;
move #sh_t6.trc/.
addq st_elt, comm_q, $1;
mv #sh_t7/, st_elt.st_name;
move #sh_t7.trc/.
addq st_elt, comm_q, $1;
mv #sh_t8/, st_elt.st_name;
move #sh_t8.trc/.
addq st_elt, comm_q, $1;
mv #sh_t9/, st_elt.st_name;
move #sh_t9.trc/.
addq st_elt, comm_q, $1;
mv #sh_t10/, st_elt.st_name;
move #sh_t10.trc/.
addq st_elt, comm_q, $1;
mv #sh_t11/, st_elt.st_name;
move #sh_t11.trc/.
addq st_elt, comm_q, $1;
mv #sh_t12/, st_elt.st_name;
move #sh_t12.trc/.
addq st_elt, comm_q, $1;
mv #sh_t13/, st_elt.st_name;
move #sh_t13.trc/.
addq st_elt, comm_q, $1;
mv #sh_t14/, st_elt.st_name;
move #sh_t14.trc/.
addq st_elt, comm_q, $1;
mv #sh_t15/, st_elt.st_name;
move #sh_t15.trc/.
addq st_elt, comm_q, $1;
mv #sh_t16/, st_elt.st_name;
move #sh_t16.trc/.
addq st_elt, comm_q, $1;
mv #sh_t17/, st_elt.st_name;
move #sh_t17.trc/.
addq st_elt, comm_q, $1;
unlock;

end;

move tasklist, proc_list;
move proc_list, br_chk;
move currtask, tempstr;
*move $1, wt_chk;

while br_chk <> $0;

delq proc_list, tcb_name;
execute tcb_name;
set tcb.stc.begin;
move wt chk, currtbc.num;
add #1, wt chk, wt chk;

if (tcb_name <> /idle/);
lock;
*type nod_name;
*type loc_comm_q, st_elt;
move tcb_name, st_elt.com;
*type $/ --LOCAL-- TASK; /
*type tcb_name;
*type $/ QUEUE; /
*typecur st_elt.st_name;
add st_elt, temp_q, $1;
unlock;

lock;
delq comm_q, st_elt;
while (st_elt.com <> nil_string);
addq st_elt, comm_q, #1;
delq comm_q, st_elt;
endwhile;
move tcb_name, st_elt.name;
*ftype #/GLOBAL-- TASK: /;
*ftype tcb_name;
*ftype # QUEUE: /;
*typenr st_elt.st_name;
addq st_elt, comm_q, #1;
unlock;
endif;
move taskstart, curr_tcb.psect;
move tcb_name, curr_tcb.name;
move #/ready/, curr_tcb.status;
move curpri, curr_tcb.priority;
move taskstk, curr_tcb.stk;
addq curr_tcb, r6r, curr_pri;
siseq precilat, br_check;
endwhile;
move temp_q, loc_comm_q;
move nil_queue, temp_q;

cntsw tempstr;
move #/rdy_tm/, rdy_str;
move #/vs_tm/, vs_str;
move #/run_tm/, run_str;
*move #/END RESPONSE ............/, end_str;
bind evt_sync, #0;
bind evt_loss_sync, #1;
bind evt_msg_rx, #2;
bind evt_signal, #3;
bind evt_clk, #4;
enable #0;
call run_rvr_task;
ENDTASK;
ENDKERNEL;
B.3.4 Application Tasks Using Blocking Kernel

<table>
<thead>
<tr>
<th>MSG Task</th>
</tr>
</thead>
</table>

STAT rdynum, integer;
STAT wtnum, integer;
STAT rntnnum, integer;

VAR byte_cnt, integer; No. of message bytes transmitted or received
VAR msg_pend, integer;
VAR msg_out, t_msg_elt;
VAR msg_in, t_msg_elt;
VAR r_msg, t_msg_elt;
VAR recv, string;
VAR takstrt, integer;
VAR takend, integer;
STAT cycle, integer;

ACTIVITY init;
*time takstrt;
move #0, msg_pend;
move #0, msg_in_type;
ENDACT;

ACTIVITY upd_cycle;
*time takend;
*sub takend, takstrt, cycle;
*time takstrt;
ENDACT;

ACTIVITY init_cnt;
move #0, byte_cnt;
move #0, msg_rdy;
ENDACT;

ACTIVITY incr_cnt;
*printer #/incr count in msg task/;
add #1, byte_cnt, byte_cnt;
ENDACT;

ACTIVITY send_data;
move #2, msg_out_type;
move #/msg/, msg_out_xor;
ENDACT;

ACTIVITY send_start;
move #1, msg_out_type;
move #/msg/, msg_out_xor;
ENDACT;

ACTIVITY send_data;
move #3, msg_out_type;
move #/msg/, msg_out_xor;
ENDACT;

ACTIVITY ack_in;
if (msg_in_type <> #0);
move #1, msg_pend;
endif;
ENDACT;

ACTIVITY clr_msg;
move #0, msg_in_type;
move #0, msg_pend;
ENDACT;

ACTIVITY set_synch;
ticks #200;
move #0, msg_in_type;
move #0, msg_pend;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprin;
SETSTATE #/clear/;
END

LOOP;
STATESELCT;

STATE clear;
IF Strue;
DO_ACT upd_cycle;
CALLP heueresp;
SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
IF (msg_pend<> #1);
CALLP reply, xov, #/msg_in/;
SETSTATE #/hendl_msg/;
ENDIF;
IF (msg_rdy<> #1);
DO_ACT init_cnt;
CALLP nilprin;
SETSTATE #/rxtx/;
ENDIF;
IF Strue;
CALLP recv, #/msg_in/, #/xov/;
SETSTATE #/sigmsg/;
ENDIF;
ENDSTATE;

STATE sigmsg;
IF Strue;
DO_ACT ack_in;
CALLP nilprin;
SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE rxtx;
IF (byte_cnt < #32);
DO_ACT incrcnt;
CALLP nilprin;
SETSTATE #/rxtx/;
ENDIF;
IF Strue;
CALLP nilprin;
SETSTATE #/msg_fin/;
ENDIF;
ENDSTATE;

STATE msg_fin;
IF (msgtype <> #1);
DO_ACT send_start;
CALLP send, #/init/,
#msg_out/, #/r_msg/;
SETSTATE #/clear/;
ENDIF;
IF (msgtype <> #2);
DO_ACT send_increment;
CALLP send, #/msg_out/,
#msg_out/, #/r_msg/;
SETSTATE #/clear/;
ENDIF;
IF (msgtype <> #3);
DO_ACT send_data;
CALLP send, #/dias/,
#msg_out/, #/r_msg/;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

STATE hendl_msg;
IF (msg_in_type <> #7);
DO_ACT clr_msg;
CALLP endresp, #1;
SETSTATE #/clear/;
ENDIF;
IF (msg_in_type <> #8);
DO_ACT set_synch;
CALLP endresp, #2;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

ENDSELECT;
ENDLOOP

379
ENDTASK;

;    CINIT Task
;
;
STAT rdy_tm, integer;
STAT wu_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_out;
VAR msg_in, t_msg_in;
VAR r_msg, t_msg;
VAR rcv, string;
VAR waiting_q, queue;
VAR msg_pending, integer;
VAR q_size, integer;
VAR start_strt, integer;
VAR end_strt, integer;
STAT cycle, integer;

ACTIVITY init:
+time start_strt;
    move #0, msg_pending;
ENDACT;

ACTIVITY upd_cycle:
+time end_strt;
+sub end_strt, start_strt, cycle;
+time start_strt;
ENDACT;

ACTIVITY get_msg:
    delq waiting_q, msg_in;
    sizeq waiting_q, q_size;
    if (q_size = #0);
        move #0, msg_pending;
    endif;
ENDACT;

ACTIVITY queue_msg:
    add msg_in, waiting_q, #1;
    move #1, msg_pending;
ENDACT;

ACTIVITY conn_req:
    move #11, msg_out.type;
    move #$cinit/, msg_out.src;
ENDACT;

ACTIVITY chnl_req:
    move #12, msg_out.type;
    move #$cinit/, msg_out.src;
ENDACT;

ACTIVITY stat_req:
    move #16, msg_out.type;
    move #$cinit/, msg_out.src;
ENDACT;

BEGIN
    DO_ACT init;
    CALLP nilprim;
    SETSTATE #$clear/;

    LOOP;
    STATESELECT;

    STATE clear:
    IF $true;
        DO_ACT
        CALLP neareq;
        SETSTATE #$strt/;
    ENDIF;
    ENDSTATE;

    STATE strt:
    IF (msg_pending = #1);
        DO_ACT
        CALLP nilprim;
        SETSTATE #$got_msg/;
    ENDIF;
    ENDSTATE;

    STATE respond:
    IF $true;
        CALLP reply, rcv, msg_out;
        SETSTATE #$got_chnl/;
    ENDIF;
    ENDSTATE;

    STATE chan_req:
    IF $true;
        CALLP send, chnl, chan/, msg_out, #r_msg/;
        SETSTATE #$got_chnl/;
    ENDIF;
    ENDSTATE;

    STATE conn_req:
    IF $true;
        CALLP send, connect/, msg_out, #r_msg/;
        SETSTATE #$got_conn/;
    ENDIF;
    ENDSTATE;

    STATE stat_req:
    IF $true;
        CALLP send, stat/, msg_out, #r_msg/;
        SETSTATE #$clear/;
    ENDIF;
    ENDSTATE;

    ENDESELECT;
ENDLOOP
ENDTASK;

;    TONES Task
;
;
STAT rdy_tm, integer;
STAT wu_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_out;
VAR msg_in, t_msg_in;
VAR r_msg, t_msg;
VAR rcv, string;
VAR waiting_q, queue;
VAR msg_pending, integer;
VAR q_size, integer;
VAR start_strt, integer;
VAR end_strt, integer;
STAT cycle, integer;

ACTIVITY init:
+time start_strt;
ENDACT;

ACTIVITY upd_cycle:
+time end_strt;
+sub end_strt, start_strt, cycle;
+time start_strt;
ENDACT;

ACTIVITY select_tone:
    ticks $350;
    if (msg_in.src = #msg/);
    move #1, msg_out.data;
    else;
    move #0, msg_out.data;
    endif;
    move #11, msg_out.type;
    move #$tones/, msg_out.src;
ENDACT;

380
BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
ENDACT;

LOOP;
STATESELECT;
  STATE clear;
    IF true;
      DO_ACT upd_cycle;
      CALLP newresp;
      SETSTATE #/start/;
    ENDIF;
  ENDSTATE;

  STATE start;
    IF true;
      CALLP recv, #/msg_in/, #/rcv/;
      SETSTATE #/respond/;
    ENDIF;
  ENDSTATE;

  STATE respond;
    IF true;
      CALLP reply, rcv, #/msg_in/;
      SETSTATE #/got_msg/;
    ENDIF;
  ENDSTATE;

  STATE got_msg;
    IF true;
      DO_ACT select_tone;
      CALLP send, #/connect/,
        #/msg_out/, #/msg/;
      SETSTATE #/clear/;
    ENDIF;
  ENDSTATE;
ENDSELECT;
ENDLOOP
ENDTASK;

; STAT Task
;
STAT rdw_tm, integer;
STAT wt_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_alt;
VAR msg_in, t_msg_alt;
VAR r_msg, t_msg_alt;
VAR rcv, string;
VAR tak_strt, integer;
VAR tak_end, integer;
STAT cycle, integer;

ACTIVITY init;
  *time tak_strt;
ENDTACT;

ACTIVITY upd_cycle;
  *time tak_end, tak_strt, cycle;
  *time tak_strt;
ENDTACT;

ACTIVITY new_call;
  tickes #200;
  update table
  move #1, stat_data;
  move #16, msg_out.type;
  move #/stat/., msg_out.src;
ENDTACT;

ACTIVITY call_fin;
  tickes #200;
  update table
  move #1, stat_data;
ENDTACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
ENDACT;

LOOP;
STATESELECT;
  STATE clear;
    IF true;
      DO_ACT upd_cycle;
      CALLP newresp;
      SETSTATE #/start/;
    ENDIF;
  ENDSTATE;

  STATE start;
    IF true;
      CALLP recv, #/msg_in/, #/rcv/;
      SETSTATE #/respond/;
    ENDIF;
  ENDSTATE;

  STATE respond;
    IF true;
      CALLP reply, rcv, #/msg_in/;
      SETSTATE #/got_msg/;
    ENDIF;
  ENDSTATE;

  STATE got_msg;
    IF (msg_in.type = #15);
      DO_ACT new_call;
      CALLP send, #/signal/,
        #/msg_out/, #/r_msg/;
      SETSTATE #/response/;
    ENDIF;
    IF (msg_in.type = #17);
      DO_ACT integ_fail;
      CALLP send, #/cancel/,
        #/msg_out/., #/r_msg/;
      SETSTATE #/clear/;
    ENDIF;
    IF true;
      DO_ACT call_fin;
      CALLP endresp, 5;
      SETSTATE #/clear/;
    ENDIF;
  ENDSTATE;

  STATE responses;
    IF true;
      CALLP endresp, 5;
      SETSTATE #/clear/;
    ENDIF;
  ENDSTATE;
ENDSELECT;
ENDLOOP
ENDTASK;

; CHMS Task
;
STAT rdw_tm, integer;
STAT wt_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_alt;
VAR msg_in, t_msg_alt;
VAR rcv, string;
VAR r_msg, t_msg_alt;
VAR waiting_q, queue;
VAR num_allocated, integer;
VAR send_mch, integer;
VAR q.size, integer;

381
VAR dest, string;
VAR tak_strt, integer;
VAR tak_end, integer;
STAT cycle, integer;
STAT mtoch, integer;
STAT cpch, integer;
STAT over_limit, integer;

ACTIVITY init;
  +time tak_strt;
ENDACT;

ACTIVITY upd_cycle;
  +time tak_end;
  +sub tak.end, tak_strt, cycle;
  +time tak_strt;
ENDACT;

ACTIVITY get_chnl;
  +if (msg.in.src = #/diags/);
    +add 1, mtocz, mtoch;
  +else;
    +add 1, cpch, cpch;
  +endif;
  if (num_allocated < #15);
    +add 1, num_allocated, num_allocated;
    move msg.in.src, dest;
    move 1, send_ack;
    move 71, msg.out.type;
    move #/chnl/., msg.out.src;
  +else;
    +add msg.in, waiting.q, #1;
    +add 1, over_limit, over_limit;
    move 0, send_ack;
  +endif;
ENDACT;

ACTIVITY free_chnl;
  sizeq waiting.q, q.size;
  if (q.size > #0);
    dq0 waiting.q, msg.in;
    move msg.in.src, dest;
    move 1, send_ack;
    move 71, msg.out.type;
    move #/chnl/., msg.out.src;
  +else;
    +num_allocated, 1, num_allocated;
    if (num_allocated < #0);
      move 0, num_allocated;
    +endif;
  +endif;
ENDACT;

ACTIVITY clear_ack;
  move 0, send_ack;
ENDACT;

BEGIN
  DO.ACT init;
  CALLP nilprim;
  SETSTATE #/clear/;
END;

LOOP;
  STATESELECT;
  STATE clear;
  IF true;
    DO.ACT upd_cycle;
    CALLP nextresp;
    SETSTATE #/start/;
  +endif;
ENDSTATE;
  STATE start;
  IF true;
    CALLP recvr, #/msg.in/., #/rcv/;
    SETSTATE #/got_msg/;
  +endif;
ENDSTATE;
  STATE get_msg;
  IF (msg.in.type = #12);
    DO.ACT get_chnl;
    CALLP nilprim;
    SETSTATE #/chk_ack/;
  ENDIF;
ENDSTATE;

STATE chk_ack;
  IF (send_ack = #1);
    DO.ACT clear_ack;
    CALLP reply, dest, #/msg_out/;
    SETSTATE #/clear/;
  ENDIF;
ENDSTATE;

STATE end;
BEGIN
  DO.ACT init;
  CALLP nilprim;
  SETSTATE #/clear/;
END;

LOOP;
  STATESELECT;
  STATE clear;
  IF true;
    DO.ACT upd_cycle;
    CALLP nextresp;
    SETSTATE #/start/;
  +endif;
ENDSTATE;
BEGIN
DO.ACT init;
CALLP nilprim;
SETSTATE #/clear/;

LOOPS;
STATESSELECT;

STATE clear;
IF #true;
DO.ACT upd_cycle;
CALLP #newresp;
STATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
IF #true;
CALLP recv, #/msg_in/, #/rsv/;
SETSTATE #/respond/;
ENDIF;
ENDSTATE;

STATE respond;
IF #true;
CALLP reply, rsv, #/msg_in/;
SETSTATE #/got_msg/;
ENDIF;
ENDSTATE;

STATE got_msg;
IF #true;
DO.ACT conn_req;
CALLP send, #/connect/, #/msg_out/, #/r_msg/;
SETSTATE #/xel_chnl/;
ENDIF;
ENDSTATE;

STATE xel_chnl;
IF #true;
DO.ACT chnl_req;
CALLP send, #/chnls/, #/msg_out/, #/r_msg/;
SETSTATE #/call_fin/;
ENDIF;
ENDSTATE;

STATE call_fin;
IF #true;
DO.ACT stat Req;
CALLP send, #/stat/, #/msg_out/, #/r_msg/;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

ENDSELECT;
ENDLOOP
ENDTASK;

; CONNECT Task
;
STAT rdy_tm, integer;
STAT wt_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_end;
VAR msg_in, t_msg_end;
VAR r_msg, t_msg_end;
VAR rsv, string;
VAR tak_strt, integer;
VAR tak_end, integer;
STAT cycle, integer;

ACTIVITY init;
* time tak_strt;
ENDACT;

ACTIVITY upd_cycle;
* time tak_end, tak_strt, cycle;
* time tak_strt;
ENDACT;

ACTIVITY make_conn;
ticks 1000;
ENDACT;

BEGIN
DO.ACT init;
CALLP nilprim;
SETSTATE #/clear/;

; SIGNAL Task
;
STAT rdy_tm, integer;
STAT wt_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_end;
VAR msg_in, t_msg_end;
VAR r_msg, t_msg_end;
VAR rsv, string;
VAR msg_pending, integer;
VAR tak_strt, integer;
VAR tak_end, integer;
STAT cycle, integer;

ACTIVITY init;
* time tak_strt;
move 00, msg_pending;
move 00, msg_in.type;
ENDACT;

ACTIVITY upd_cycle;
move 00, msg_in.type;
* time tak_end, tak_strt, cycle;
* time tak_strt;
ENDACT;

ACTIVITY init_sent;
ticks 750;
ENDACT;

ACTIVITY upd_integ;
ticks 75;
ENDACT;

ACTIVITY send_stat;
move $17, mag_out.type;
move $19, mag_out.type;
move $11, mag_out.type;
ENACT;

ACTIVITY fin_stat;
move $19, mag_out.type;
move $11, mag_out.type;
ENDACT;

ACTIVITY send_diag;
move $3, mag_out.type;
move $1, signal, mag_out.src;
ENDACT;

ACTIVITY ask_in;
if (mag_in.type < $10);
*print $rcv = /;
*printc $rcv;
mov $1, mag_pending;
endif;
ENDACT;

ACTIVITY clr_upd_integ;
mov $10, mag_in.type;
ticks $16;
mov $10, mag_pending;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
LOOP:
STATESELECT:

STATE clear;
IF #true;
DO_ACT upd_cycle;
CALLP nrwexp;
SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
IF (mag_pending = $1);
CALLP reply, rcv, #/mag_in/;
SETSTATE #/hdl1_mag/;
ENDIF;
IF (sig_rdy = $1);
DO_ACT init_cnt;
CALLP nilprim;
SETSTATE #/rcv/;
ENDIF;
IF #true;
CALLP recv, #/mag_in/,
SETSTATE #/sigmag/;
ENDIF;
ENDSTATE;

STATE sigmag;
IF #true;
DO_ACT ask_in;
CALLP nilprim;
SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE rx:
IF (sigtype = "$54")
DO_ACT upd_integ;
CALLP nilprim;
SETSTATE #/clear/;
ENDIF;
IF (sigtype = "$05")
DO_ACT send_xstat;
CALLP send, #/stat/,
SETSTATE #/mag_out/,
ENDIF;
IF (sigtype = "$60")
DO_ACT fin_stat;
SENDSTATE #/stat/,
ENDSTATE;

STATE diag;
IF #true;
DO_ACT send_diags;
CALLP send, #/diags/,
SETSTATE #/mag_out/,
ENDIF;
ENDIF;
ENDSTATE;

STATE hndl_mag;
IF #true;
DO_ACT clr_upd_integ;
CALLP nilprim;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

ENDSELECT;
ENDLOOP
ENDTASK;

| DIAG Task |
| STAT rdy_tm, integer; | STAT wt_tm, integer; | STAT run_tm, integer; |
| VAR mag_out, t_mag_out; | VAR mag_in, t_mag_in; | VAR x_mag, t_mag_x; |
| VAR rcv, string; | VAR waiting_q, queue; | VAR mag_pending, integer; |
| VAR q_size, integer; | VAR dest, string; | VAR dlg_num, integer; |
| VAR tak_bsr, integer; | VAR tak_end, integer; | STAT cycle, integer; |
| ACTIVITY init;
*time tak_bsr;
ENDACT;

ACTIVITY upd_cycle;
*time tak_end;
*sub tak_end, tak_bsr, cycle;
*time tak_bsr;
ENDACT;

ACTIVITY get_mag;
deq waiting_q, mag_in;
deq waiting_q, q_size;
if (q_size = $0);
mov $0, mag_pending;
endif;
ENDACT;

ACTIVITY queue_mag;
add mag_in, waiting_q, $1;
mov $1, mag_pending;
ENDACT;

ACTIVITY chnl_req;
mov #$12, mag_out.type;
mov #/diags/, mag_out.src;
ENDACT;

ACTIVITY choose_diag;
rand $4, dlg_num;
mov $21, mag_out.type;
mov #/diags/, mag_out.src;
if (dest = $0);
mov #/ndiags/, dest;
else;
  if (dest = #1);
    move #/chnl_req/, dest;
  else;
    if (dest = #2);
      move #/msgdiag/, dest;
    else;
      move #/error/, dest;
    endif;
  endif;
endif;
ENDACT;

BEGIN
DO_ACT init;
  CALLP nilprim;
  SETSTATE #/clear/;
ENDACT;

LOOP:
  STATESELECT;
  STATE clear;
    IF #true;
      DO_ACT upd_cycle;
      CALLP nreply;
      SETSTATE #/start/;
    ENDIF;
  ENDSTATE;
  STATE start;
    IF (msg_pending = #1);
      DO_ACT get_msg;
      CALLP nilprim;
      SETSTATE #/get_msg/;
    ENDIF;
    IF #true;
      CALLP recv, #/msg_in/,
      SETSTATE #/respond/;
    ENDIF;
  ENDSTATE;
  STATE respond;
    IF #true;
      CALLP reply, rcv, #/msg_out/;
    ENDIF;
  ENDSTATE;
  STATE get_msg;
    IF #true;
      DO_ACT chnl_req;
      CALLP send, #/chnl idle/,
      SETSTATE #/get_chnl/;
    ENDIF;
  ENDSTATE;
  STATE get_chnl;
    IF #true;
      DO_ACT chnl_req;
      CALLP send, #/chnl idle/,
      SETSTATE #/pick_diag/;
    ENDIF;
  ENDSTATE;
  STATE pick_diag;
    IF #true;
      DO_ACT choose_diag;
      CALLP send, dest,
      SETSTATE #/clear/;
    ENDIF;
  ENDSTATE;
  ENDSSELECT;
ENDLOOP
ENDTASK;
STATE get_mag;
    IF (mag.in.type = #23);
        DO.ACT = do_scan;
        CALLP nilprim;
    SETSTATE #/chk_results/;
ENDIF;
ENDSTATE;

STATE chk_results;
    IF (fail = #1);
        DO.ACT = report_fail;
        CALLP send, #/fault/,
             #/mag.out/, #/r_mag/;
        SETSTATE #/responses/;
        ENDIF;
    IF (true);
        CALLP nilprim;
        SETSTATE #/responses/;
    ENDIF;
ENDSTATE;

STATE responses;
    IF (fail = #0);
        CALLP endresp, #7;
        SETSTATE #/clear/;
    ENDIF;
    IF (true);
        CALLP nilprim;
        SETSTATE #/clear/;
    ENDIF;
ENDSTATE;

ENDSELECT;
ENDLOOP;
ENDTASK;

; AUDITS Task
;
STAT rdyl_tm, integer;
STAT rtyl_tm, integer;
STAT runl_tm, integer;
VAR msg_out, t_mag_out;
VAR r_mag, t_mag_out;
VAR r_col, string;
VAR aud1_cnt, integer;
VAR aud2_cnt, integer;
VAR aud3_cnt, integer;
VAR task_strt, integer;
VAR task_end, integer;
STAT cycle, integer;

ACTIVITY init;
    +time task_strt;
ENDACT;

ACTIVITY upd_cycle;
    +time task_end, task_strt, cycle;
    +time task_strt;
ENDACT;

ACTIVITY incr_cnt;
    add #1, aud1_cnt, aud1_cnt;
    add #1, aud2_cnt, aud2_cnt;
    add #1, aud3_cnt, aud3_cnt;
ENDACT;

ACTIVITY tell_time;
    move #23, msg_out.type;
    move #/advises/, msg_out.src;
ENDACT;

ACTIVITY audi;
    move #0, aud1_cnt;
    move #10, msg_out.type;
    move #/advises/, msg_out.src;
ENDACT;

ACTIVITY aud2;
    move #0, aud2_cnt;
    move #10, msg_out.type;
    move #/advises/, msg_out.src;
ENDACT;

ACTIVITY aud3;
    move #0, aud3_cnt;
    move #10, msg_out.type;
    move #/advises/, msg_out.src;
ENDACT;

BEGIN
    DO.ACT = init;
    CALLP nilprim;
    SETSTATE #/clear/;
    LOOP;
        STATESELECT;
            STATE clear;
                IF (true);
                    DO.ACT = upd_cycle;
                    CALLP newresp;
                    SETSTATE #/start/;
                ENDIF;
            ENDSTATE;
            STATE start;
                IF (true);
                    DO.ACT = incr_cnt;
                    CALLP sleep;
                    SETSTATE #/trigger_time/;
                ENDIF;
            ENDSTATE;
            STATE trigger_time;
                IF (true);
                    DO.ACT = tell_time;
                    CALLP send, #/time/,
                          #/msg_out/, #/r_mag/;
                    SETSTATE #/chk_audi/;
                ENDIF;
            ENDSTATE;
            STATE chk_audi;
                IF (aud1_cnt = #10);
                    DO.ACT = audi;
                    CALLP send, #/diag/,
                          #/msg_out/, #/r_mag/;
                    SETSTATE #/chk_audi2/;
                ENDIF;
                IF (true);
                    CALLP nilprim;
                    SETSTATE #/chk_audi2/;
                ENDIF;
            ENDSTATE;
            STATE chk_audi2;
                IF (aud2_cnt = #20);
                    DO.ACT = aud2;
                    CALLP send, #/diag/,
                          #/msg_out/, #/r_mag/;
                    SETSTATE #/chk_audi3/;
                ENDIF;
                IF (true);
                    CALLP nilprim;
                    SETSTATE #/chk_audi3/;
                ENDIF;
            ENDSTATE;
            STATE chk_audi3;
                IF (aud3_cnt = #30);
                    DO.ACT = aud3;
                    CALLP send, #/diag/,
                          #/msg_out/, #/r_mag/;
                    SETSTATE #/clear/;
                ENDIF;
        ENDSTATE;
    ENDLOOP;
ENDSTATE;
IF $true;
CALLP nilprim;
STATE #/clear/;
ENDIF;
ENDSTATE;

ENDESELECT;
ENDLOOP
ENDTASK;

; TIDIG Task
; STAT rdy_tm, integer;
STAT wt_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_end;
VAR msg_in, t_msg_end;
VAR r_msg, t_msg_end;
VAR rcv, string;
VAR result, integer;
VAR fail, integer;
VAR tek_strt, integer;
VAR tek_end, integer;
STAT cycle, integer;

ACTIVITY init;
*time tek_strt;
ENDACT;

ACTIVITY upd_cycle;
*time tek_end;
*sub tek_end, tek_strt, cycle;
*time tek_strt;
ENDACT;

ACTIVITY stop_mtc;
move $21, msg_out.type;
move $/tidig/, msg_out.arc;
ENDACT;

ACTIVITY do_diag;
move $0, fail;
ticks $400;
rand $10, result;
if (result == $0)
mov $1, fail;
endif;
ENDACT;

ACTIVITY report_fail;
mov $10, msg_out.type;
mov $/tidig/, msg_out.arc;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
STATE #/clear/;
LOOP;
STATESSELECT;
STATE clear;
IF $true;
DO_ACT
CALLP noresp;
STATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
IF $true;
CALLP rcv, #/msg_in/, #/rcv/;
STATE #/response/;
ENDIF;
ENDSTATE;

STATE respond;
IF $true;
CALLP reply, recv, #/msg_in/;
STATE got_msg/;
ENDIF;
ENDSTATE;

STATE got_msg;
IF $true;
DO_ACT
send, #/signal/, #/msg_out/, #/r_msg/;
STATE #/diagose/;
ENDIF;
ENDSTATE;

STATE diagnose;
IF $true;
DO_ACT
do_diagn;
CALLP nilprim;
STATE #/chk_result/
ENDIF;
ENDSTATE;

STATE chk_result;
IF (fail == $1);
DO_ACT
report_fail;
CALLP send, #/fault/, #/msg_out/, #/r_msg/;
STATE #/response/;
ENDIF;
IF $true;
CALLP nilprim;
STATE #/response/;
ENDIF;
ENDSTATE;

STATE respond;
IF (fail == $0);
CALLP endresp, #/;
STATE #/clear/;
ENDIF;
IF $true;
CALLP nilprim;
STATE #/clear/;
ENDIF;
ENDSTATE;

ENDESELECT;
ENDLOOP
ENDTASK;

; MSGDIAG Task
; STAT rdy_tm, integer;
STAT wt_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_end;
VAR msg_in, t_msg_end;
VAR r_msg, t_msg_end;
VAR rcv, string;
VAR result, integer;
VAR fail, integer;
VAR tek_strt, integer;
VAR tek_end, integer;
STAT cycle, integer;

ACTIVITY init;
*time tek_strt;
ENDACT;

ACTIVITY upd_cycle;
*time tek_end;
*sub tek_end, tek_strt, cycle;
*time tek_strt;
ENDACT;

ACTIVITY stop_mtc;
mov $20, msg_out.type;
move $/msgdiag/, msg_out.src;
ENDACT;

ACTIVITY do_diag;
move $0, fail;
ticks $200;
rand $10, result;
if (result = $81);
move $1, fail;
endif;
ENDACT;

ACTIVITY report_fail;
move $10, msg_out.type;
move $/msgdiag/, msg_out.src;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
ENDSTATE;

LOOP:
STATESSELECT;

STATE clear;
IF $true;
DO_ACT upd_cycle;
        CALLP newresp;
        SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
IF $true;
        CALLP rcv, #/msg_in/, #/rcv/;
        SETSTATE #/response/;
ENDIF;
ENDSTATE;

STATE respond;
IF $true;
        CALLP reply, rcv, #/msg_in/;
        SETSTATE #/got_msg/;
ENDIF;
ENDSTATE;

STATE got_msg;
IF $true;
        DO_ACT stop_integ;
        CALLP send, #/signal/,$/msg_out/, $/x_msg/;
        SETSTATE #/diagnose/;
ENDIF;
ENDSTATE;

STATE diagnose;
IF $true;
        DO_ACT nilprim;
        SETSTATE #/chk_results/;
ENDIF;
ENDSTATE;

STATE chk_results;
IF (fail = $1);
        DO_ACT report_fail;
        CALLP send, #/fail/,$/msg_out/, $/x_msg/;
        SETSTATE #/response/;
ENDIF;

IF $true;
        CALLP nilprim;
        SETSTATE #/response/;
ENDIF;
ENDSTATE;

STATE respond;
IF (fail = $0);
        DO_ACT endresp, $/6;
        SETSTATE #/clear/;
ENDIF;

IF $true;
        CALLP nilprim;
ENDSTATE;

STATE clear;
IF $true;
        DO_ACT upd_cycle;
        CALLP newresp;
        SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
IF $true;
        CALLP rcv, #/msg_in/, #/rcv/;
        SETSTATE #/response/;
ENDIF;
ENDSTATE;

STATE respond;
IF $true;
        CALLP reply, rcv, #/msg_in/;
ENDSTATE;

STATESSELECT;

STATE clear;
ENDIF;
ENDSTATE;

ENDSELECT;
ENDLOOP
ENDSTATE;

; TONEDIAG Task
;
STAT rdy_tm, integer;
STAT wt_tm, integer;
STAT run_tm, integer;
VAR msg_out, x_msg.int;
VAR msg_in, x_msg.int;
VAR x_msg, t_msg.int;
VAR rcv, string;
VAR result, integer;
VAR fail, integer;
VAR task_strt, integer;
VAR task_end, integer;
STAT cycle, integer;

ACTIVITY init;
etime task_strt;
ENDACT;

ACTIVITY upd_cycle;
etime task_end;
etime task_end, task_strt, cycle;
etime task_strt;
ENDACT;

ACTIVITY select_tone;
mov $2, msg_out.type;
mov $/tonedias/, msg_out.src;
ENDACT;

ACTIVITY do_diagnostic;
mov $0, fail;
ticks $300;
rand $10, result;
if (result = $81);
mov $1, fail;
endif;
ENDACT;

ACTIVITY report_fail;
mov $10, msg_out.type;
mov $/tonedias/, msg_out.src;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
ENDSTATE;

LOOP:
STATESSELECT;

STATE clear;
IF $true;
DO_ACT upd_cycle;
        CALLP newresp;
        SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
IF $true;
        CALLP rcv, #/msg_in/, #/rcv/;
        SETSTATE #/response/;
ENDIF;
ENDSTATE;

STATE respond;
IF $true;
        CALLP reply, rcv, #/msg_in/;
        SETSTATE #/got_msg/;
ENDIF;
ENDSTATE;

STATE got_msg;
IF $true;
        DO_ACT stop_integ;
        CALLP send, #/signal/,$/msg_out/, $/x_msg/;
        SETSTATE #/diagnose/;
ENDIF;
ENDSTATE;

STATE diagnose;
IF $true;
        DO_ACT nilprim;
        SETSTATE #/chk_results/;
ENDIF;
ENDSTATE;

STATE chk_results;
IF (fail = $1);
        DO_ACT report_fail;
        CALLP send, #/fail/,$/msg_out/, $/x_msg/;
        SETSTATE #/response/;
ENDIF;

IF $true;
        CALLP nilprim;
        SETSTATE #/response/;
ENDIF;
ENDSTATE;

STATE respond;
IF (fail = $0);
        DO_ACT endresp, $/6;
        SETSTATE #/clear/;
ENDIF;

IF $true;
        CALLP nilprim;
ENDSTATE;
SETSTATE #/got_msg/;
ENDIF;
ENDSTATE;
STATE got_msg;
IF #true;
DO_ACT select_tone;
CALLP send, #/tone/,
#/#mag.out/, #/r_msg/;
STATE done;
#diagnose/;
ENDIF;
ENDSTATE;
STATE done;
ENDIF;
ENDSTATE;
STATE chk_results;
IF #true;
DO_ACT do_diag;
CALLP nilprim;
STATE done;
#chk_results/;
ENDIF;
ENDSTATE;
STATE done;
ENDIF;
ENDSTATE;
STATE responding;
IF (fail = #0);
DO_ACT report.fail;
CALLP send, #/fault/,
#/#mag.out/, #/r_msg/;
STATE done;
#response/;
ENDIF;
ENDSTATE;
STATE done;
ENDIF;
ENDSTATE;
STATE respond;
IF (fail = #0);
CALLP endresp, #5;
STATE done;
#response/;
ENDIF;
ENDSTATE;
STATE done;
ENDIF;
ENDSTATE;
ENDSELECT;
ENDLOOP
ENDTASK;

; CHNLDIAG Task
;
STAT rdv_t, integer;
STAT wt_t, integer;
STAT run_t, integer;
VAR mag.out, t_msg.alt;
VAR mag.in, t_msg.alt;
VAR r_msg, t_msg.alt;
VAR rcv, string;
VAR result, integer;
VAR fail, integer;
VAR tsk_start, integer;
VAR tsk_end, integer;
STAT cycle, integer;

ACTIVITY init;
+time tsk_start;
ENDACT;
ACTIVITY upd_cycle;
+time tsk_end;
+time tsk_start, cycle;
ENDACT;
ACTIVITY connect_chnl;
move #1, mag.out.type;
move #/chnldiag/, mag.out.src;
ENDACT;
ACTIVITY do_diag;
move #0, fail;
ticks #150;
rand #10, result;
if (result = #90)
move #1, fail;
endif;
ENDACT;
ACTIVITY report_fail;
move #10, mag.out.type;
move #/chnldiag/, mag.out.src;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
STATE done;
#clear/;
LOOP:
STATE SELECT:
STATE clear;
IF #true;
DO_ACT upd_cycle;
CALLP neresp;
STATE done;
#start/;
ENDIF;
ENDSTATE;
STATE start;
IF #true;
CALLP receive, #/mag.in/, #/rcv/;
STATE done;
#respond/;
ENDIF;
ENDSTATE;
STATE respond;
IF #true;
CALLP reply, rcv, #/mag.in/;
STATE done;
#got_msg/;
ENDIF;
ENDSTATE;
STATE got_msg;
IF #true;
DO_ACT connect_chnl;
CALLP send, #/connect/,
#/#mag.out/, #/r_msg/;
STATE done;
#signal/;
ENDIF;
ENDSTATE;
STATE signal;
IF #true;
DO_ACT stop_integ;
CALLP send, #/signal/,
#/#mag.out/, #/r_msg/;
STATE done;
#diagnose/;
ENDIF;
ENDSTATE;
STATE diag;
ENDSTATE;
STATE done;
ENDIF;
ENDSTATE;
STATE done;
ENDIF;
ENDSTATE;
STATE done;
ENDIF;
ENDSTATE;
STATE tsk_start, integer;
STATE tsk_end, integer;
STATE cycle, integer;

ACTIVITY init;
+time tsk_start;
ENDACT;
ACTIVITY upd_cycle;
+time tsk_end;
+time tsk_start, cycle;
+time tsk_end;
ENDACT;
ACTIVITY connect_chnl;
move #11, mag.out.type;
move #/chnldiag/, mag.out.src;
ENDACT;
ACTIVITY stop_integ;
SETSTATE #/response/
ENDIF;
ENDSTATE;
STATE response;
IF (call = 0); endresp, #S;
CALLP
SETSTATE #/clear/;
ENDIF;
IF $true; CALLP nilprim;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;
ENDESELECT;
ENDLOOP
ENDTASK;
B.3.5 Semaphore Based Kernel

KERNEL CODE FILE

This kernel contains the following primitives:

wait
signal
init semaphore
sleep
get data from another task
hilprim

The following external message types are supported:

$1 - start call setup
$2 - provide tone
$3 - run diagnostic
$4 - integrity ok
$5 - integrity failure
$6 - call takedown

The following internal message types are supported:

$7 - START - request call setup
$8 - TONES - request for tone
$9 - DIAG - request single random diagnostic
$10 -
$11 -
$12 - MSG - report fault
$13 - MSG - perform and report synch operation
$14 - FAULT - synch loss
$15 - FAULT - general fault report
$16 - CONNECT - perform connect/disconnect
$17 - CHELS - allocate channel
$18 - CHELS - free channel
$19 - all tasks - channel granted
$20 - STAT - call setup complete
$21 - SIGNAL - start integrity checking
$22 - STAT - integrity failure report
$23 - END - caller hung up
$24 - STAT - call finished
$25 - STAT - stop integrity checking
$26 - all diag - run diagnostic
$27 - TIMTC - stop carrier checking
$28 - TIMTC - monitor carriers

The following response types are supported:

$1 - MGQ - fault report sent to switching center
$2 - MSG - synch source changed
$3 - MSG - switching center tone request connected
$4 - STAT - call finished
$5 - STAT - call started
$6 - MGQ - diagnostic completed without failure
$7 - TIMTC - carrier scan completed without failure

KERNEL base:

TASK kern_task, 80;

TYPE t_tob_state =
    rdy_time : integer *;
    run_time : integer *;
    ut_time : integer *;
    begin : integer *;
    finish : integer *;
    resp_q : queue ;
    of t_trace_elt

TYPE t_shared_state = data_trace : queue ;
    of t_trace_elt

TYPE t_tob =
    name : string *;
    task name
    nnn : integer *;
    status : string *;
    priority : integer *;
    pnt : integer *;
    program counter
   _elt_str: string * ;
    rcv msg elt
    stk : stack *
    local task stack
    sta : t_tob_state;
    task control block

TYPE t_id_struct =
    nom : string *;
    trc_name : string *;
    st_name : string;
SHARED sh_mag, t_sh_data;
SHARED sh_clinit, t_sh_data;
SHARED sh_ones, t_sh_data;
SHARED sh_chklen, t_sh_data;
SHARED sh_connect, t_sh_data;
SHARED sh_signal, t_sh_data;
SHARED sh_default, t_sh_data;
SHARED sh_diag, t_sh_data;
SHARED tt_data, integer;
SHARED stat_data, integer;
SHARED num_pending, integer;

STAT sends, integer;
STAT os_ovhd, integer;
STAT switches, integer;
STAT m_n_wait, integer;
STAT m_n_signal, integer;
STAT tt_wait, integer;
STAT tt_signal, integer;
STAT evt_0, integer;
STAT evt_1, integer;
STAT evt_2, integer;
STAT evt_3, integer;
STAT evt_4, integer;
VAR start_os, integer;
VAR end_os, integer;
VAR elapsed, integer;
VAR rdy_str, string;
VAR wt_str, string;
VAR run_str, string;
VAR end_str, string;

VAR local, integer;
VAR loc_comm_q, queue;  
VAR loc_task, integer;
VAR sh_comm_q, queue;
VAR taskm, string;
VAR temp2_q, queue;
VAR wt_el, t_id, struct;
VAR q2, size, integer;
VAR g_found, integer;
VAR g_q, size, integer;

VAR wait, queue;
VAR semtr, string;
VAR cnt, integer;
VAR wt_chk, integer;
VAR mag_str, string;
VAR dest_str, string;
VAR temp_tcb, t_tcb;
VAR ret_po, integer;
VAR temp_stk, stack;
VAR found, integer;
VAR data_size, integer;
VAR nil_queues, queue;
VAR nil_str, string;
VAR m_num, integer;
VAR done, integer;

; Global Kernel Variables
VAR curr_tcb, t_tcb  
VAR proc_list, queue  
VAR br_chk, integer  
VAR temp_str, string;  
VAR tcb_name, string;  
VAR rtx, queue  
VAR q_qsize, integer  
VAR temp_q, queue

; VARIABLE declarations for procedure RUN_RTR_TASK
VAR error, boolean;  
Set if no ready processes

; VARIABLE declarations for procedure SCHED_RTR_TASK
VAR rtr_pcb, t_tcb;  
PCB to be scheduled
PROCEDURE run_rtr_task

This procedure takes the first ready to run task off the rtr queue (this is the one with the highest priority) and copies its processor state into the current processor state. If there are no ready to run tasks, then an error message is printed and the kernel terminates.

PROC run_rtr_task;
    sizeq rtr, br_chk;
    IF (br_chk = 90);
        printc "$ERROR: NO RTR PROCESSES/";
        halt;
    ELSE;
        delq rtr, curr_tcb;
        move curr_tcb.name, tcb_name;
        curre tcb_tcb_name;
        add 1, switches, switches;
        *time curr_tcb.sts.finish;
        *sub curr_tcb.sts.finish, curr_tcb.sts.begin, curr_tcb.sts.begin;
        *add curr_tcb.sts.begin, curr_tcb.sts.rdy_time, curr_tcb.sts.rdy_time;
        *move #/rdy_tw, rdy_str;
        *move curr_tcb.sts.rdy_time, &rdy_str;
        *move curr_tcb.sts.wt_time, &wt_str;
        *time curr_tcb.sts.begin;
        move #/running/, curr_tcb.status;
        move curr_tcb.stk, stack;
        *time end_os;
        *sub end_os, start_os, elapsed;
        *add elapsed, os_ovhd, os_ovhd;
        enable $10;
        move curr_tcb.pcnt, pc;
    ENDBIF;
ENDPROC;

PROCEDURE sched_rtr_task (rtr_tcb : t_tcb)

This procedure accepts a process control block and adds it to the ready to run queue in a prioritised fashion.

PROC sched_rtr_task;
    pop taskstk, rtr_tcb;
    move #/ready/, rtr_tcb.status;
    *time rtr_tcb.sts.finish;
    *sub rtr_tcb.sts.finish, rtr_tcb.sts.begin, rtr_tcb.sts.begin;
    *add rtr_tcb.sts.begin, rtr_tcb.sts.wt_time, rtr_tcb.sts.wt_time;
    *move taskstk, tmpstk;
    *time rtr_tcb.sts.begin;
    addq rtr_tcb, rtr, rtr_tcb.priority;
    IF (rtr_tcb.priority > curr_tcb.priority);
        pop taskstk, ret_tcb;
        pop taskstk, ret_tcb;
        move ret_tcb, curr_tcb.pcnt;
        move taskstk, curr_tcb.stk;
        *time curr_tcb.sts.finish;
        *sub curr_tcb.sts.finish, curr_tcb.sts.begin, curr_tcb.sts.begin;
        *add curr_tcb.sts.begin, curr_tcb.sts.run_time, curr_tcb.sts.run_time;
        *move curr_tcb.sts.run_time, &run_str;
        *time curr_tcb.sts.begin;
        move #/ready/, curr_tcb.status;
        addq curr_tcb, rtr, curr_tcb.priority;
        call run_rtr_task;
        halt;
    ENDBIF;
ENDPROC;

PROCEDURE get_local (semname : string)

This procedure accepts a semaphore name and sets the variable local to 1 if it is resident on the local node.

PROC get_local;
    pop taskstk, tempstk;
    move 90, g_found;
    sizeq loc_comm_q, g_q_size;
    while (g_q_size > 90) & (g_found = 90);
delq loc_comm_q, st_elt;
if (st_elt.st_name = tempstr) | (st_elt.trc_name = tempstr);
move $1, g_found;
endif;
addq st_elt, loc_comm_q, $1;
sub g_q.size, $1, g_q.size;
endwhile;
move g_found, local;
ENDPROC;

; PROCEDURE get_local_task (taskname : string)
; This procedure accepts a task name and sets the variable local_task
to 1 if it is resident on the local node.
PROC get_loc_task;
  pop taskstat, tempstr;
  move tasklist, proclist;
  move $0, g_found;
  aseq proclist, g_q.size;
  while (g_q.size > $0) & (g_found = $0);
    delq proclist, tasknm;
    if (tasknm = tempstr);
      move $1, g_found;
    endif;
    sub g_q.size, $1, g_q.size;
  endwhile;
  move g_found, loc_task;
ENDPROC;

; PRIMITIVE nilprim
; This primitive simply ensures consistency of semaphores across processors.
PRIM nilprim;
  disable $10;
  if (num_pending > $0);
    move $0, found;
    aseq sh_comm_q, q.size;
    while (q.size > $0) & (found = $0);
      delq sh_comm_q, st_elt;
      move st_elt.st_name, semptr;
      lock;
      aseq @semaphore.wait_q, q2.size;
      if (@semaphore.count > $0) & (q2.size > $0);
        move @semaphore.wait_q, temp2_q;
        delq temp2_q, temp_tcb;
        move nil_queue, temp2_q;
        call get_loc_task, temp_tcb.name;
        if (loc_task = $1);
          sub @semaphore.count, $1, @semaphore.count;
          move $1, found;
          delq @semaphore.wait_q, temp_tcb;
        endif;
      else;
        move st_elt.trc_name, semptr;
        aseq @semaphore.wait_q, q.size;
        if (@semaphore.count > $0) & (q2.size > $0);
          move @semaphore.wait_q, temp2_q;
          delq temp2_q, temp_tcb;
          move nil_queue, temp2_q;
          call get_loc_task, temp_tcb.name;
          if (loc_task = $1);
            sub @semaphore.count, $1, @semaphore.count;
            move $1, found;
            delq @semaphore.wait_q, temp_tcb;
          endif;
        endif;
      unlock;
    endif;
    addq st_elt, sh_comm_q, $1;
    sub q.size, $1, q.size;
  endwhile;
  if (found = $1);
    sub num_pending, $1, num_pending;
  endif;
end if;

*time start_cs;
call sched_rtr_task, temp_tcb;
endif;
endif;
enable $10;
ENDPRIM;

: PRIMITIVE wait (Gsemtr : t_sema)

; This primitive performs a semaphore wait operation.
PRIM wait:
disable $10;
*time start_cs;
*add $1, nn_wait, nn_wait;
pop taskstk, ssemtr;
lock;

if (!Gsemtr.count = $0);
pop taskstk, ret_pc;
move #wait, curr_tcb.status;
move ret_pc, curr_tcb.pcnt;
move taskstk, curr_tcb.wavk;
add curr_tcb, Gsemtr.wait_q, $1;
unlock;
*time end_cs;
*sub end_cs, start_os, elapsed;
*add elapsed, tt_wait, tt_wait;
*add elapsed, os_ovhd, os_ovhd;
*time curr_tcb.sts.finish;
*sub curr_tcb.sts.finish, curr_tcb.sts.begin, curr_tcb.sts.run_time;
*add curr_tcb.sts.begin, curr_tcb.sts.run_time, curr_tcb.sts.run_time;
*time curr_tcb.sts.begin;
call run_rtr_task;
else;
*sub Gsemtr.count,$1, Gsemtr.count;
*time end_cs;
*sub end_cs, start_os, elapsed;
*add elapsed, tt_wait, tt_wait;
*add elapsed, os_ovhd, os_ovhd;
endif;
unlock;
enable $10;
ENDPRIM;

: PRIMITIVE signal (Gsemtr : t_sema)

; This primitive performs a semaphore signal operation.
PRIM signal:
disable $10;
*time start_cs;
*add $1, nn_signal, nn_signal;
pop taskstk, ssemtr;
call get_local, ssemtr;
lock;
siuch Gsemtr.wait_q, br_chk;
if ((br_chk = $0) | (local <> $1));
add $1, Gsemtr.count, Gsemtr.count;
if (local <> $1);
add $1, num_pending, num_pending;
endif;
else;
add Gsemtr.wait_q, temp_tcb;
unlock;
call sched_rtr_task, temp_tcb;
endif;
unlock;

*time end_cs;
*sub end_cs, start_os, elapsed;
*add elapsed, tt_signal, tt_signal;
*add elapsed, os_ovhd, os_ovhd;
enable $10;
ENDPRIM;

: PRIMITIVE initsem (Gsemtr : t_sema; cnt : integer)

; This primitive performs a semaphore initialization operation.
PRIM initm;
  disable $10;
  pop taskstk, cnt;
  pop taskstk, semptr;
  move cnt, &sempt.r.count;
  move nil_queue, &semptr.wait_q;
  enable $10;
ENDPRIM;

; PRIMITIVE sleep
;
; This primitive simply blocks the calling task by putting it in
; the sleep wait queue.
PRIM sleep;
  disable $10;
  *time start_os;
  pop taskstk, ret_pc;
  *time curr_tcb.sts.finish;
  *sub curr_tcb.sts.finish, curr_tcb.sts.begin, curr_tcb.sts.begin;
  *add curr_tcb.sts.begin, curr_tcb.sts.run_time, curr_tcb.sts.run_time;
  move curr_tcb.sts.run_time, &run_str;
  *time curr_tcb.sts.begin;
  move #/wait, curr_tcb.status;
  move taskstk, curr_tcb.stk;
  move ret_pc, curr_tcb.pcnt;
  addq curr_tcb, wait, curr_tcb.priority;
  call run_str_task;
ENDPRIM;

; PRIMITIVE get_data (task_name : string; data_name : string; size : integer)
;
; This primitive simulates data access between tasks. If the
; specified source task is not resident on the processor, then a number
; of shared accesses are made. One access is made for every 100 bytes
; of data requested.
PRIM get_data;
  disable $10;
  *time start_os;
  pop taskstk, data_size;
  pop taskstk, msg_str;
  pop taskstk, dest_str;
  call get_loc_task, dest_str;
  if (loc_task = $0);
    while (data_size > $0);
      lock;
      move &msg_str, q_size;
      unlock;
      sub data_size, $100, data_size;
    endwhile;
  else;
    while (data_size > $0);
      move q_size, q_size;
      sub data_size, $100, data_size;
    endwhile;
  endif;
  *time end_os;
  *sub end_os, start_os, elapsed;
  *add elapsed, os_ovhd, os_ovhd;
  enable $10;
ENDPRIM;

; PRIMITIVE endResp
;
; This primitive simply calls response update to print the
; response traces.
PRIM endResp;
  disable $10;
  *pop taskstk, m_num;
  *type $/response completed. Type : /;
  *type m_num;
  *time m_num;
  *type $/ Time : /;
  *type m_num;
; enable $10;
; ENSPRIM;

; PRIMITIVE new.resp
; This primitive clears the calling task's current trace queue.
PRIM newresp;
; ENSPRIM;

; PRIMITIVE evt_synch
; This primitive serves as the interrupt handler for the synchronization
; primitive. This interrupt is raised periodically every 5 seconds.
PRIM evt_synch;
disable $10;
ackint $0;
*add $1, evt.O, evt.O;
*time start_os;
*iseq await, wt_chk;
move $0, found;
while ((wt_chk > $0) & (~found));
delq await, temp_tcb;
if (temp_tcb.name = $/synch/);
move $1, found;
else;
addq temp_tcb, await, temp_tcb.priority;
sub wt_chk, #1, wt_chk;
endif;
endwhile;
if (found);
*type $/synch event (0) at time : /;
*typeusr start_os;
call sched rtn_task, temp_tcb;
endif;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_overhd, os_overhd;
enable $10;
; ENSPRIM;

; PRIMITIVE evt_loss_synch
; This primitive serves as the interrupt handler for the synchronization
; loss primitive. This interrupt is raised sporadically to signal a loss of
; synchronization.
PRIM evt_loss_synch;
disable $10;
ackint $1;
*add $1, evt.i, evt.i;
*time start_os;
*iseq await, wt_chk;
move $0, found;
while ((wt_chk > $0) & (~found));
delq await, temp_tcb;
if (temp_tcb.name = $/synch/);
move $1, synch_loss;
move $1, found;
else;
addq temp_tcb, await, temp_tcb.priority;
sub wt_chk, #1, wt_chk;
endif;
endwhile;
if (found);
*type $/synch loss (1) at time : /;
*typeusr start_os;
call sched rtn_task, temp_tcb;
endif;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_overhd, os_overhd;
enable $10;
; ENSPRIM;

; PRIMITIVE evt_msg_rx
;
This primitive serves as the interrupt handler for the incoming external message interrupt. This interrupt is raised sporadically to signal the arrival of a message. It makes the MSU task ready to run from the blocked receive state.

PRIMITIVE evt_msg;
  disable #10;
  select #2;
  add #1, evt_2, evt_2;
  time start_os;
  sizeq sh_msg.avail.wait_q, done;
  if (done > #0);
    move #1, msg_rdy;
    rand #10, ut_chk;
    if (ut_chk < #5);
      move #1, msgtype;
    else;
      if (ut_chk < #8);
        move #2, msgtype;
      else;
        move #3, msgtype;
      endif;
    endif;
  endif;
  type #/msg event (2) of type : /
  type msgtype;
  type #/ at time : /
  typopc start_os;
  delq sh_msg.avail.wait_q, temp_tcb;
  call schedctxtask, temp_tcb;
  time end_os;
  sub end_os, start_os, elapsed;
  add elapsed, os_ovhd, os_ovhd;
  enable #10;
ENDPRIMITIVE;

PRIMITIVE evt_signal
  This primitive serves as the interrupt handler for the signalling interrupt handler. This interrupt is raised sporadically to signal the arrival of a signalling event. It makes the SIGNAL task ready to run from the blocked receive state.

PRIMITIVE evt_signal;
  disable #10;
  select #3;
  add #1, evt_3, evt_3;
  time start_os;
  sizeq sh_signal.avail.wait_q, done;
  if (done > #0);
    move #1, sig_rdy;
    rand #10, ut_chk;
    if (ut_chk < #5);
      move #8, sigtype;
    else;
      if (ut_chk < #8);
        move #6, sigtype;
      else;
        move #5, sigtype;
      endif;
    endif;
  endif;
  type #/sig event (3) of type : /
  type sigtype;
  type #/ at time : /
  typopc start_os;
  delq sh_signal.avail.wait_q, temp_tcb;
  call schedctxtask, temp_tcb;
  time end_os;
  sub end_os, start_os, elapsed;
  add elapsed, os_ovhd, os_ovhd;
  enable #10;
ENDPRIMITIVE;

PRIMITIVE evt_clk
  This primitive serves as the interrupt handler for the clock interrupt used as a timing base for the audit task. This is a periodic interrupt.
PRIM evt_clk;
disable #10;
schedule #4;
+add #1, evt_clk, #5, evt_.4;
time start_clk;
sizeq svait, ut_clk;
move #0, found;
while ((ut_clk > #0) & ('"found' ));
delq svait, temp_tcb;
if (temp_tcb.name = \"/audits/\" );
  move #1, found;
else;
  addq temp_tcb, svait, temp_tcb.priority;
  sub ut_clk, #1, ut_clk;
  endif;
endwhile;

if (found);
  *type \#audit event (4) at time : /
  *type os start_os;
  call sched.rtr_task., temp_tcb;
  time end_os;
  +sub end_os, start_os, elapsed;
  +add elapsed, os_ovhd, os_ovhd;
  enable #10;
ENDPRI;

BEGIN
move \#/nodes1/., nod_name;
disable #10;
lock;
if (nod_name = \#/nodes1/);
  move \#/sh Mag.mutex/., st_elt.st_name;
  move \#/sh Mag.avail/., st_elt_tcb_name;
  addq st_elt, sh_comm_q, #1;
  addq st_elt, sh_comm_q, #1;
  move \#/sh fault.mutex/., st_elt.st_name;
  move \#/sh fault.avail/., st_elt_tcb_name;
  addq st_elt, sh_comm_q, #1;
  move \#/sh diag.mutex/., st_elt.st_name;
  move \#/sh diag.avail/., st_elt_tcb_name;
  addq st_elt, sh_comm_q, #1;
  move \#/sh cinit.mutex/., st_elt.st_name;
  move \#/sh cinit.avail/., st_elt_tcb_name;
  addq st_elt, sh_comm_q, #1;
  move \#/sh tones.mutex/., st_elt.st_name;
  move \#/sh tones.avail/., st_elt_tcb_name;
  addq st_elt, sh_comm_q, #1;
  move \#/sh chnl.mutex/., st_elt.st_name;
  move \#/sh chnl.avail/., st_elt_tcb_name;
  addq st_elt, sh_comm_q, #1;
  move \#/sh connect.mutex/., st_elt.st_name;
  move \#/sh connect.avail/., st_elt_tcb_name;
  addq st_elt, sh_comm_q, #1;
  move \#/sh signal.mutex/., st_elt.st_name;
  move \#/sh signal.avail/., st_elt_tcb_name;
  addq st_elt, sh_comm_q, #1;
  call initmsg, \#/sh Mag.mutex/., #1;
  disable #10;
endif;
if (nod_name = \#/nodes2/);
  move \#/sh synch.mutex/., st_elt.st_name;
  move \#/sh synch.avail/., st_elt_tcb_name;
  addq st_elt, loc_comm_q, #1;
  move \#/sh fault.mutex/., st_elt.st_name;
  move \#/sh fault.avail/., st_elt_tcb_name;
  addq st_elt, loc_comm_q, #1;
  move \#/sh audits.mutex/., st_elt.st_name;
  move \#/sh audits.avail/., st_elt_tcb_name;
  addq st elt, loc_comm_q, #1;
  move \#/sh timed.mutex/., st_elt.st_name;
  move \#/sh timed.avail/., st_elt_tcb_name;
  addq st elt, loc_comm_q, #1;
ENDBEG;
move #/sh_diag淼mutex/, st_elt.st_name;
move #/sh_diag淼avail/, st_elt.troc_name;
addq st_elt, loc_comm_q, #1;
addq st_elt, sh淼comm_q, #1;
move #/sh_tid淼mutex/, st_elt.st_name;
move #/sh_tid淼avail/, st_elt.troc_name;
addq st_elt, loc_comm_q, #1;
move #/sh_mag淼mutex/, st_elt.st_name;
move #/sh_mag淼avail/, st_elt.troc_name;
addq st_elt, loc_comm_q, #1;
move #/sh_cinit淼mutex/, st_elt.st_name;
move #/sh_cinit淼avail/, st_elt.troc_name;
addq st_elt, sh淼comm_q, #1;
move #/sh_tones淼mutex/, st_elt.st_name;
move #/sh_tones淼avail/, st_elt.troc_name;
addq st_elt, sh淼comm_q, #1;
move #/sh_ch淼mutex/, st_elt.st_name;
move #/sh_ch淼avail/, st_elt.troc_name;
addq st_elt, sh淼comm_q, #1;
move #/sh淼mutex/, st_elt.st_name;
move #/sh淼avail/, st_elt.troc_name;
addq st_elt, sh淼comm_q, #1;
call init淼item, #/sh淼synch淼mutex/, #1;
disable #10;
call init淼item, #/sh淼fault淼mutex/, #1;
disable #10;
call init淼item, #/sh淼audits淼mutex/, #1;
disable #10;
call init淼item, #/sh淼time淼mutex/, #1;
disable #10;
call init淼item, #/sh淼diag淼mutex/, #1;
disable #10;
call init淼item, #/sh淼tid淼mutex/, #1;
disable #10;
call init淼item, #/sh淼mag淼mutex/, #1;
disable #10;
call init淼item, #/sh淼tones淼mutex/, #1;
disable #10;
call init淼item, #/sh淼ch淼mutex/, #1;
disable #10;
endif;

if (nod淼name = #/node82/);
move #/sh淼cinit淼mutex/, st_elt.st_name;
move #/sh淼cinit淼avail/, st_elt.troc_name;
addq st_elt, loc_comm_q, #1;
addq st_elt, sh淼comm_q, #1;
move #/sh淼tones淼mutex/, st_elt.st_name;
move #/sh淼tones淼avail/, st_elt.troc_name;
addq st_elt, loc_comm_q, #1;
addq st_elt, sh淼comm_q, #1;
move #/sh淼stat淼mutex/, st_elt.st_name;
move #/sh淼stat淼avail/, st_elt.troc_name;
addq st_elt, loc_comm_q, #1;
move #/sh淼ch淼mutex/, st_elt.st_name;
move #/sh淼ch淼avail/, st_elt.troc_name;
addq st_elt, loc_comm_q, #1;
addq st_elt, sh淼comm_q, #1;
move #/sh淼end淼mutex/, st_elt.st_name;
move #/sh淼end淼avail/, st_elt.troc_name;
addq st_elt, loc_comm_q, #1;
move #/sh淼connect淼mutex/, st_elt.st_name;
move #/sh淼connect淼avail/, st_elt.troc_name;
addq st_elt, loc_comm_q, #1;
addq st_elt, sh淼comm_q, #1;
move #/sh淼signal淼mutex/, st_elt.st_name;
move #/sh淼signal淼avail/, st_elt.troc_name;
addq st_elt, loc_comm_q, #1;
move #/sh淼sync淼mutex/, st_elt.st_name;
move #/sh_signal.avail/, st_elts.trc.name;
addq st_elts.local_comm_q, #1;
addq st_elts.sh_comm_q, #1;
move #/sh_mag.mutex/, st_elts.st_name;
move #/sh_mag.avail/, st_elts.trc.name;
addq st_elts.sh_comm_q, #1;
move #/sh_fault.mutex/, st_elts.st_name;
move #/sh_fault.avail/, st_elts.trc.name;
addq st_elts.sh_comm_q, #1;
move #/sh_diag.mutex/, st_elts.st_name;
move #/sh_diag.avail/, st_elts.trc.name;
addq st_elts.sh_comm_q, #1;
call init., #/sh_cinit.mutex/, #1;
disable #10;
call init., #/sh_c tones.mutex/, #1;
disable #10;
call init., #/sh_stat.mutex/, #1;
disable #10;
call init., #/sh_chnl.s.mutex/, #1;
disable #10;
call init., #/sh_end.mutex/, #1;
disable #10;
call init., #/sh_connect.mutex/, #1;
disable #10;
call init., #/sh_signal.mutex/, #1;
disable #10;
endif;

UNLOCK;

move tasklist, proclist;
sizeq proclist, br_ckh;
move currtask, tempstr;
move #1, wt_ckh;
while br_ckh > #0;
delq proclist, tcb_name;
ctime curr_tcb.sta.begin;
move wt_ckh, curr_tcb.num;
add #1, wt_ckh, wt_ckh;
move taskstr, curr_tcb.pmt;
move tcb_name, curr_tcb.name;
move #ready, curr_tcb.status;
move currprio, curr_tcb.priority;
move taskstr, curr_tcb.stk;
addq curr_tcb, ptx, currprio;
sizeq proclist, br_ckh;
endwhile;

cstrftime tempstr;
move #/dt(tm), rdy_str;
move #/st_tm/, wt(str);
move #/run_tm/, run_str;
move #/END RESPONSE ............./, end_str;
bind evt_sync, #0;
bind evt_lost_sync, #1;
bind evt_mag refunds, #2;
bind evt_signal, #3;
bind evt_clk, #4;
call run.rwtask;

ENDTASK;
ENDERNL;
B.3.6 Semaphore Based Application Tasks

```assembly
BEGIN
  DO_ACT Init;
  CALLP nilprim;
  SETSTATE #/clear/;
  LOOP;
  STATESELECT;
    STATE clear;
    IF (msg.pending = #1);
      CALLP nilprim;
      SETSTATE #/handl_msg/;
    ENDIF;
    ENDSTATE;
    STATE start;
    IF (msg.pending = #1);
      CALLP nilprim;
      SETSTATE #/start/;
    ENDIF;
    ENDSTATE;
    STATE rdy_mutex;
    IF (#true);
      CALLP unit, #/sh_msg.mute/;
      SETSTATE #/signas/;
    ENDIF;
    ENDSTATE;
    STATE inocr_mutex;
    IF (#true);
      CALLP unit, #/sh_msg.mute/;
      SETSTATE #/start/;
    ENDIF;
    ENDSTATE;
    STATE xtxtx;
    IF (byte_cnt < #32);
      DO_ACT incr_cnt;
      CALLP nilprim;
      SETSTATE #/xtxtx/;
    ENDIF;
    ENDSTATE;
    STATE msg_fir;
    IF (msgtype = #1);
      CALLP wait, #/sh_mtime.mute/;
      SETSTATE #/msg_fir/;
    ENDIF;
    ENDSTATE;
    STATE msg_fir2;
    IF (msgtype = #2);
      CALLP wait, #/sh_mtime.mute/;
      SETSTATE #/msg_fir2/;
    ENDIF;
    ENDSTATE;
    STATE msg_fir3;
    IF (msgtype = #3);
      CALLP wait, #/sh_mtime.mute/;
      SETSTATE #/msg_fir3/;
    ENDIF;
    ENDSTATE;
  ENDSTATE;
  ACT send_tone;
    move $2, msg.out.type;
    move $#msg, msg.out.arc;
    addq msg.out, sh_tone.mages, $1;
  ENDACT;
  ACT send_start;
    move $1, msg.out.type;
    move $#msg, msg.out.arc;
    addq msg.out, sh_init.mages, $1;
  ENDACT;
  ACT send_diag;
    move $3, msg.out.type;
    move $#msg, msg.out.arc;
    addq msg.out, sh_diag.mages, $1;
  ENDACT;
  ACT sck_in;
    streq sh msg.mages, q.size;
    if (q.size <= #0);
      delq sh msg.mages, msg_in;
      move $1, msg.pending;
      endif;
  ENDACT;
  ACT sl_next;
    move $0, msg_in.type;
    move $0, msg.pending;
  ENDACT;
  ACT set_syncs;
    ticks #300;
    move $0, msg_in.type;
    move $0, msg.pending;
  ENDACT;
END
```
CALLP signal, #/sh_tones.mutex/;
ENDIT;

STATE msg_fin3;
IF (msgtype = #3);
    CALLP signal, #/sh_diag.avail/;
    SETSTATE #/msg_fin3/;
ENDIF;
ENDSTATE;

STATE hnd1_msg;
IF ((msg_in.type = #7));
    DO.ACT clr_msg;
    CALLP endresp, #1;
    SETSTATE #/clear/;
ENDIF;
IF (msgtype = #3);
    CALLP signal, #/sh_diag.avail/;
    SETSTATE #/clear/;
ENDIF;
ENDSTATE;

TOMES Task

VAR msg_out, t_mag_alt;
VAR msg_in, t_mag_alt;
VAR r_mag, t_mag_alt;
VAR count, string;
VAR tak_strt, integer;
VAR tak_end, integer;
STAT cycle, integer;

ACTIVITY init;
    stime tak_strt;
ENDACT;

ACTIVITY upd_cycle;
    stime tak_end;
    sub tak_end, tak_strt, cycle;
    stime tak_strt;
ENDACT;

ACTIVITY select_tone;
    ticks #350;
    if (msg_in.src = #/mag/);
        move #1, msg_out.data;
    else;
        move #0, msg_out.data;
    endif;
    move #11, msg_out.type;
    move #/tones, msg_out.src;
ENDACT;

ACTIVITY copy_msg;
    delq sh_tones.mages, msg_in;
ENDACT;

Brazil

BEGIN
    DO.ACT init;
    CALLP nullpim;
    SETSTATE #/clear/;

LOOP
    STATE_SELECT;
    IF #true;
        DO.ACT upd_cycle;
        CALLP neresp;
        SETSTATE #/start/;
    ENDIF;
    ENDSSTATE;
    IF #true;
        CALLP wait, #/sh_tones.avail/;
        SETSTATE #/repl/;
    ENDIF;
    ENDSSTATE;
    IF #true;
        CALLP wait, #/sh_tones.avail/;
        SETSTATE #/repl2/;
    ENDIF;
    ENDSSTATE;
    IF #true;
        DO.ACT copy_msg;
        CALLP signal, #/sh_tones.mutex/;
        SETSTATE #/got_msg/;
    ENDIF;
    ENDSSTATE;
    IF #true;
        DO.ACT select_tone;
        CALLP wait, #/sh_connect.mutex/;
        SETSTATE #/got_mag2/;
    ENDIF;
    ENDSSTATE;
    IF #true;
        DO.ACT copy_out;
        CALLP signal, #/sh_connect.mutex/;
        SETSTATE #/got_mag3/;
    ENDIF;
    ENDSSTATE;
    IF #true;
        CALLP signal, #/sh_connect.avail/;
        SETSTATE #/clear/;
    ENDIF;
    ENDSSTATE;
ENDLOOP
ENDTASK;
VAR rev, string;
VAR tak_strt, integer;
VAR tak_end, integer;
STAT cycle, integer;

ACTIVITY init;
  setime tak_strt;
ENDACT;

ACTIVITY upd_cycle;
  setime tak_end;
  sub tak_end, tak_strt, cycle;
  setime tak_strt;
ENDACT;

ACTIVITY copy_in;
  delq sh_end.mages, msg_in;
ENDACT;

ACTIVITY chnl_req;
  move $13, msg_out.type;
  move $/end/, msg_out.arc;
  addq msg_out, sh_chlns.mages, $1;
ENDACT;

ACTIVITY conn_req;
  move $11, msg_out.type;
  move $/end/, msg_out.arc;
  addq msg_out, sh_connect.mages, $1;
ENDACT;

ACTIVITY stat_req;
  move $19, msg_out.type;
  move $/end/, msg_out.arc;
  addq msg_out, sh_connect.mages, $1;
ENDACT;

BEGIN
  DU_ACT init;
  CALLP nilprim;
  SETSTATE #$clear/;
  LOOP;
  STATESSELECT;

  STATE clear;
  IF $true;
    DU_ACT upd_cycle;
    CALLP newreq;
    SETSTATE #$start/;
  ENDIF;
  ENDSTATE;

  STATE start;
  IF $true;
    CALLP wait, $/sh_end.avai/;
    SETSTATE #$resp1/;
  ENDIF;
  ENDSTATE;

  STATE resp1;
  IF $true;
    CALLP wait, $/sh_end.avai/;
    SETSTATE #$resp2/;
  ENDIF;
  ENDSTATE;

  STATE resp2;
  IF $true;
    DU_ACT copy_in;
    CALLP signal, $/sh_end.avai/;
    SETSTATE #$got_msg/;
  ENDIF;
  ENDSTATE;

  STATE got_msg;
  IF $true;
    CALLP wait, $/sh_connect.avai/;
    SETSTATE #$got_msg2/;
  ENDIF;
  ENDSTATE;

  STATE got_msg2;
  IF $true;
    DU_ACT conn_req;
    CALLP signal, $/sh_connect.avai/;
    SETSTATE #$got_msg3/;
  ENDIF;
  ENDSTATE;

  STATE got_msg3;
  IF $true;
    CALLP signal, $/sh_connect.avai/;
    SETSTATE #$rel_chnl/;
  ENDIF;
  ENDSTATE;

  STATE rel_chnl;
  IF $true;
    CALLP wait, $/sh_chlns.avai/;
    SETSTATE #$rel_ch2/;
  ENDIF;
  ENDSTATE;

  STATE rel_ch2;
  IF $true;
    CALLP chnl_req;
    CALLP signal, $/sh_chlns.avai/;
    SETSTATE #$rel_ch3/;
  ENDIF;
  ENDSTATE;

  STATE rel_ch3;
  IF $true;
    CALLP signal, $/sh_chlns.avai/;
    SETSTATE #$call_fin/;
  ENDIF;
  ENDSTATE;

  STATE call_fin;
  IF $true;
    CALLP wait, $/sh_stat.avai/;
    SETSTATE #$call_f2/;
  ENDIF;
  ENDSTATE;

  STATE call_f2;
  IF $true;
    DU_ACT stat_req;
    CALLP signal, $/sh_stat.avai/;
    SETSTATE #$call_f3/;
  ENDIF;
  ENDSTATE;

  STATE call_f3;
  IF $true;
    CALLP signal, $/sh_stat.avai/;
    SETSTATE #$clear/;
  ENDIF;
  ENDSTATE;

  ENDLOOP;
  ENDSTATE;

405
\begin{verbatim}
| CINIT Task |
| STAT rd_y, integer;
| STAT w1, integer;
| STAT run, integer;

| VAR msg, out, t_msg, elp;
| VAR msg, in, t_msg, elp;
| VAR r_msg, t_msg, elp;
| VAR wait, waiting, q, queue;
| VAR msg, pending, integer;
| VAR q, size, integer;
| VAR task, str_t, integer;
| VAR task, end, integer;
| STAT cycle, integer;

| ACTIVITY init: |
| time task, start;
| move w1, msg, pending;

| ACTIVITY upd_cycle: |
| time task, end;
| sub task, end, task, start, cycle;
| time task, start;

| ACTIVITY get_msg:
| dcall waiting, q, msg, in;
| elseq waiting, q, q, size;
| if (q, size = 0);
| move w0, msg, pending;
| endif;

| ACTIVITY queue_msg:
| addq msg, in, waiting, q, $1;
| move $1, msg, pending;

| ACTIVITY copy_msg:
| dcall sh_connect, messages, msg, in;

| ACTIVITY conn req:
| move $11, msg, out, type;
| move #sinit, msg, out, src;
| addq msg, out, sh_connect, messages, $1;

| ACTIVITY chnl req:
| move $12, msg, out, type;
| move #sinit, msg, out, src;
| addq msg, out, sh_connect, messages, $1;

| ACTIVITY stat req:
| move $16, msg, out, type;
| move #sinit, msg, out, src;
| addq msg, out, sh_start, messages, $1;

| BEGIN |

| DO ACT Init;
| callpnilprim;
| setstate #clear/1;

| LOOP |

| STATES SELECT: |

| STATE clear:
| if #true;
| do ACT upd_cycle;
| callp newresp;
| setstate #/str/2;
| endif;

| STATE str_t:
| if (msg, pending = #1);

| DO ACT get_msg:
| callp nilprim;
| setstate #/got_msg/1;

| ENDIF |

| STATE resp1:
| if #true;
| callp wait, #/sinit, available;
| setstate #/resp1/1;
| endif;

| ENDSTATE;

| STATE resp2;
| if #true;
| do ACT copy_msg;
| callp signal, #/sinit, mutex/1;
| setstate #/got_msg/1;
| endif;

| ENDSTATE;

| STATE got_msg1:
| if #true;
| callp wait, #/sinit, mutex/1;
| setstate #/got_msg2/1;
| endif;

| ENDSTATE;

| STATE got_msg2:
| if #true;
| do ACT chnl, req;
| callp signal, #/sinit, mutex/1;
| setstate #/got_msg1/1;
| endif;

| ENDSTATE;

| STATE got_msg3:
| if #true;
| callp signal, #/sinit, available;
| setstate #/vt, ch/1;
| endif;

| ENDSTATE;

| STATE vt, ch1:
| if #true;
| callp wait, #/sinit, available;
| setstate #/vt, ch2/1;
| endif;

| ENDSTATE;

| STATE vt, ch2:
| if #true;
| callp wait, #/sinit, mutex/1;
| setstate #/vt, ch3/1;
| endif;

| ENDSTATE;

| STATE vt, ch3:
| if #true;
| do ACT copy_msg;
| callp signal, #/sinit, mutex/1;
| setstate #/chlk, msg/1;
| endif;

| ENDSTATE;

| STATE chlk, msg:
| if (msg, in, type = #11);
| callp nilprim;
| setstate #/got, ch/1;
| endif;

| ENDIF |

| STATE quot:
| if #true;
| do ACT queue_msg;
| callp nilprim;
| setstate #/vt, ch/1;
| endif;

| ENDSTATE;

| STATE got, ch:
| if #true;
| callp wait, #/sinit, mutex/1;
\end{verbatim}
SETSTATE #/got_ch2/;
ENDIF;
ENDSTATE;

STATE got_ch2;
IF #true;
DO_ACT
conn_req;
CALLP signal, #/sh_connect_mutex/;
SETSTATE #/got_ch3/;
ENDIF;
ENDSTATE;

STATE got_ch3;
IF #true;
CALLP signal, #/sh_connect_avail/;
SETSTATE #/got_cnf/;
ENDIF;
ENDSTATE;

STATE got_cnf;
IF #true;
CALLP wait, #/sh_stat_mutex/;
SETSTATE #/got_cn2/;
ENDIF;
ENDSTATE;

STATE got_cn2;
IF #true;
DO_ACT
stat_req;
CALLP signal, #/sh_stat_mutex/;
SETSTATE #/got_cn3/;
ENDIF;
ENDSTATE;

STATE got_cn3;
IF #true;
CALLP signal, #/sh_stat_avail/;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

ENDSELECT;
ENDLOOP
ENDSTATE;

; STAT Task

STAT rdy_tm, integer;
STAT wt_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_elt;
VAR msg_in, t_msg_elt;
VAR r_msg, t_msg_elt;
VAR rev, string;
VAR task_strt, integer;
VAR task_end, integer;
STAT cycle, integer;

ACTIVITY init;
etime task_strt;
ENDACT;

ACTIVITY upd_cycle;
time task_end;
sub task_end, task_strt, cycle;
time task_strt;
ENDACT;

ACTIVITY msg_call;
ticks #200;
update table
move #1, stat_data;
mov #10, msg_out_type;
mov #1, stat_data;
move #stat/., msg_out/src/;
addq msg_out, sh_fault.mages, #1;
ENDACT;

ACTIVITY copy_in;
deq sh_stat.mages, msg_in;
ENDACT;

ACTIVITY call_zin;
ticks #200;
mov #1, stat_data;
ENDACT;

ACTIVITY intan_fail;
mov #10, msg_out_type;
mov #1, stat_data;
mov #stat/., msg_out/src/;
addq msg_out, sh_fault.mages, #1;
ENDACT;

BEGIN
DO_ACT
init;
CALLP nilprim;
SETSTATE #/clear/;
LOOP;
STATE_SELECT;

STATE clear;
IF #true;
DO_ACT
upd_cycle;
CALLP nextesp;
SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
IF #true;
CALLP wait, #/sh_stat_avail/;
SETSTATE #/resp1/;
ENDIF;
ENDSTATE;

STATE resp1;
IF #true;
CALLP wait, #/sh_stat_mutex/;
SETSTATE #/resp2/;
ENDIF;
ENDSTATE;

STATE resp2;
IF #true;
DO_ACT
copy_in;
CALLP signal, #/sh_stat_mutex/;
SETSTATE #/got_msg/;
ENDIF;
ENDSTATE;

STATE got_msg;
IF (msg_in.type = #15);
CALLP wait, #/sh_signal_mutex/;
SETSTATE #/got_msg2/;
ENDIF;

STATE got_msg2;
IF (msg_in.type = #17);
CALLP wait, #/sh_fault_mutex/;
SETSTATE #/got_msg3/;
ENDIF;

STATE got_msg3;
IF (msg_in.type = #18);
DO_ACT
new_call;
CALLP signal, #/sh_signal_mutex/;
SETSTATE #/got_msg5/;
ENDIF;

STATE got_msg5;
ENDSTATE;

ENDSTATE;
STATE got_msg3;
  IF (msg.in.type = #16);
    CALLP signal, #/sh_signal.avail/;
    SETSTATE #/response/;
  ENDIF;
  IF (msg.in.type = #17);
    CALLP signal, #/sh_fault.avail/;
    SETSTATE #/clear/;
  ENDIF;
  IF (true);
    CALLP nilprim;
    SETSTATE #/clear/;
  ENDIF;
ENDSTATE;

STATE response;
  IF (#true);
    CALLP endresp, #5;
    SETSTATE #/clear/;
  ENDIF;
ENDSTATE;

ENDSELECT;
ENDDROP
ENDTASK;

| CHNLs Task |
|            |
STAT rdy_tm, integer;
STAT ut_tm, integer;
STAT run_tm, integer;

VAR msg_out, t_msg_wait;
VAR msg_in, t_msg_wait;
VAR rcv, string;
VAR r_msg, t_msg_wait;
VAR waiting_q, queue;
VAR num_allocated, integer;
VAR send_ack, integer;
VAR q_size, integer;
VAR dest1, string;
VAR dest2, string;
VAR dest3, string;

VAR task_strt, integer;
VAR task_end, integer;
STAT clycle, integer;
STAT mtoch, integer;
STAT cpch, integer;
STAT over_limit, integer;

ACTIVITY init;
  *time task_strt;
ENDACT;

ACTIVITY upd_cycle;
  *time task_end;
  *sub task_end, task_strt, cycle;
  *time task_strt;
ENDACT;

ACTIVITY copy_in;
  dl.q sh.chnl.s.mages, msg_in;
ENDACT;

ACTIVITY copy_out;
  addq msg_out, @dest3, #1;
ENDACT;

ACTIVITY get_chnl;
  IF (msg.in.type = #/diags/);
    *add #1, mtoch, mtoch;
    move #/sh_diags.mtxr, dest1;
    move #/sh_diags.avail, dest2;
    move #/sh_diags.mages, dest3;
  ELSE;
    *add #1, cpch, cpch;
    move #/sh.chnls.mtxr, dest1;
    move #/sh.chnls.avail, dest2;
  ENDIF;
ENDACT;

STATE got_msg2;
  IF (msg.in.type = #12);
    DO.ACT copy_in;
    CALLP signal, #/sh.chnls.mtxr/;
    SETSTATE #/got_msg/;
  ENDIF;
ENDSTATE;

STATE cpy_msg2;
  IF (#true);
    DO.ACT cpy_msg;
    CALLP signal, #/sh.chnls.mtxr/;
    SETSTATE #/got_msg/;
  ENDIF;
ENDSTATE;

STATE cpy_msg3;
  IF (#true);
    DO.ACT cpy_msg;
    CALLP signal, #/sh.chnls.mtxr/;
    SETSTATE #/got_msg/;
  ENDIF;
ENDSTATE;

STATE got_msg;
  IF (num_allocated < #16);
    add #1, num_allocated, num_allocated;
    move #1, send_ack;
    move #/4, mag_out.type;
    move #/ohnbs/, mag_out.src;
  ELSE;
    addq mag_in, waiting_q, #1;
    add #1, over_limit, over_limit;
    move #0, send_ack;
  ENDIF;
ENDACT;

ACTIVITY free_chnl;
  sizeq waiting_q, q_size;
  IF (q_size > #0);
    dl.q waiting_q, msg_in;
    IF (msg.in.src = #/diags/);
      move #/sh_diags.mtxr, dest1;
      move #/sh_diags.avail, dest2;
      move #/sh_diags.mages, dest3;
    ELSE;
      move #/sh.chnls.mtxr, dest1;
      move #/sh.chnls.avail, dest2;
      move #/sh.chnls.mages, dest3;
    ENDIF;
    move #1, send_ack;
    move #/4, mag_out.type;
    move #/ohnbs/, mag_out.src;
  ELSE;
    sub num_allocated, #1, num_allocated;
    IF (num_allocated < #0);
      move #0, num_allocated;
    ENDIF;
  ENDIF;
ENDACT;

ACTIVITY clear_ack;
  move #0, send_ack;
ENDACT;

BEGIN
DO.ACT init;
  CALLP nilprim;
  SETSTATE #/clear/;
ENDACT;

ACTIVITY clear_ack;
  move #0, send_ack;
ENDACT;

STATE clear;
  IF (#true);
    DO.ACT upd_cycle;
    CALLP nresp;
    SETSTATE #/start/;
  ENDIF;
ENDSTATE;

STATE start;
  IF (#true);
    CALLP wait, #/sh.chnls.avail/;
    SETSTATE #/cpy_msg/;
  ENDIF;
ENDSTATE;

STATE cpy_msg;
  IF (#true);
    CALLP wait, #/sh.chnls.mtxr/;
    SETSTATE #/cpy_msg2/;
  ENDIF;
ENDSTATE;

STATE cpy_msg2;
  IF (#true);
    DO.ACT copy_in;
    CALLP signal, #/sh.chnls.mtxr/;
    SETSTATE #/got_msg/;
  ENDIF;
ENDSTATE;

STATE got_msg;
  IF (msg.in.type = #12);
    DO.ACT get_chnl;
    CALLP nilprim;
CONNECT Task

| STAT rdy_cm, integer;
| STAT wt_cm, integer;
| STAT run_cm, integer;
| VAR mag_in, t_mag_out;
| VAR r_mag, t_mag_out;
| VAR rcv, string;
| VAR tak_start, integer;
| VAR tak_end, integer;
| STAT cycle, integer;

ACTIVITY init;
  @time tak_start;
ENDACT;

ACTIVITY upd_cycle;
  @time tak_end;
  @sub tak_end, tak_start, cycle;
  @time tak_start;
ENDACT;

ACTIVITY make_conn;
  ticks 5000;
ENDACT;

ACTIVITY copy_in;
  delq sh_connectmsgs, mag_in;
ENDACT;

BEGIN
  DO.ACT init;
  CALLP nilprim;
  SETSTATE #clear/;
ENDLOOP;

STATE clear;
  IF #true;
  DO.ACT upd_cycle;
  CALLP
  SETSTATE #start/;
ENDIF;

ENDSTATE;

STATE start;
  IF #true;
  CALLP
  SETSTATE #res-background/;
ENDIF;

ENDSTATE;

STATE res-background;
  IF #true;
  CALLP
  SETSTATE #resp2/;
ENDIF;

ENDSTATE;

STATE resp2;
  IF #true;
  DO.ACT copy_in;
  CALLP
  SETSTATE #got_msg/;
ENDIF;

ENDSTATE;

STATE got_msg;
  IF ((mag_in.src = #/tones/) &
      (mag_in.data = #1));
  DO.ACT "make_conn";
  CALLP endresp, $3;
  SETSTATE #clear/;
ENDIF;

ENDSTATE;

STATE endresp;
  IF #true;
  DO.ACT nilprim;
  CALLP nilprim;
  SETSTATE #clear/;
ENDIF;

ENDSTATE;

ENDSELECT;
ENDTASK;

SIGNAL Task

| STAT rdy_cm, integer;
| STAT wt_cm, integer;
| STAT run_cm, integer;
| VAR mag_out, t_mag_out;
| VAR r_mag, t_mag_out;
| VAR rcv, string;
| VAR mag_pending, integer;
| VAR tak_start, integer;
| VAR tak_end, integer;
| STAT cycle, integer;

ACTIVITY init;
  @time tak_start;
ENDACT;

ACTIVITY upd_cycle;
  @time tak_end;
  @sub tak_end, tak_start, cycle;
  @time tak_start;
ENDACT;

ACTIVITY make_conn;
  ticks 5000;
ENDACT;

ACTIVITY copy_in;
  delq sh_connectmsgs, mag_in;
ENDACT;

BEGIN
  DO.ACT init;
  CALLP nilprim;
  SETSTATE #clear/;
ENDLOOP;

STATE clear;
  IF #true;
  DO.ACT upd_cycle;
  CALLP
  SETSTATE #start/;
ENDIF;

ENDSTATE;

STATE start;
  IF #true;
  CALLP
  SETSTATE #res-background/;
ENDIF;

ENDSTATE;

STATE res-background;
  IF #true;
  CALLP
  SETSTATE #resp2/;
ENDIF;

ENDSTATE;

STATE resp2;
  IF #true;
  DO.ACT copy_in;
  CALLP
  SETSTATE #got_msg/;
ENDIF;

ENDSTATE;

STATE got_msg;
  IF ((mag_in.src = #/tones/) &
      (mag_in.data = #1));
  DO.ACT "make_conn";
  CALLP endresp, $3;
  SETSTATE #clear/;
ENDIF;

ENDSTATE;

STATE endresp;
  IF #true;
  DO.ACT nilprim;
  CALLP nilprim;
  SETSTATE #clear/;
ENDIF;

ENDSTATE;

ENDSELECT;
ENDTASK;
move #17, msg_out.type;
move #/signal/, msg_out.src;
addq msg_out, sh_stat.mages, #1
ENDACT;

ACTIVITY fin_stat;
move #19, msg_out.type;
move #/signal/, msg_out.src;
addq msg_out, sh_stat.mages, #1
ENDACT;

ACTIVITY send_diag;
move #3, mag_out.type;
move #/signal/, msg_out.src;
addq msg_out, sh_diags.mages, #1
ENDACT;

ACTIVITY ack_in;
sizeq sh_signal.mages, q_size;
if (q_size > 81);
delq sh_signal.mages, mag_in;
move #1, mag_pending;
endif;
ENDACT;

ACTIVITY clr_upd_integ;
mov w0, mag_in.type;
ticks #76;
mov w0, mag_pending;
ENDACT;

BEGIN
DO_ACT init;
call nilprim;
SETSTATE #/clear/;
END;

STATESELECT;

STATE clear;
if #true;
DO_ACT upd_cycle;
call nilprim;
SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
if (mag_pending = #1);
call nilprim;
SETSTATE #/hndl_msg/;
ENDIF;
if (sig_rdy = #1);
DO_ACT init_cnt;
call nilprim;
SETSTATE #/tx/;
ENDIF;
if #true;
call wait, #/sh_signal.avail/;
SETSTATE #/mt_mux/;
ENDIF;
ENDSTATE;

STATE wt_mux;
if #true;
call wait, #/sh_signal.avail/;
SETSTATE #/msgw/;
ENDIF;
ENDSTATE;

STATE msgw;
if #true;
DO_ACT ack_in;
call nilprim;
SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE rx;
if (sig_type = #4);
DO_ACT upd_integ;
call nilprim;
SETSTATE #/clear/;
ENDIF;

IF #true;
call wait, #/sh_stat.mux/;
SETSTATE #/tx2/;
ENDIF;
ENDSTATE;

STATE tx2;
if (sig_type = #6);
DO_ACT send_stat;
call nilprim;
SETSTATE #/tx3/;
ENDIF;
if (sig_type = #6);
DO_ACT fin_stat;
call nilprim;
SETSTATE #/tx5/;
ENDIF;
ENDSTATE;

STATE tx3;
if (sig_type = #6);
call nilprim;
SETSTATE #/diag/;
ENDIF;
if (sig_type = #6);
call nilprim;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

STATE diag;
if #true;
call wait, #/sh_diags.mux/;
SETSTATE #/diag2/;
ENDIF;
ENDSTATE;

STATE diag2;
if #true;
DO_ACT send_diag;
call nilprim;
SETSTATE #/diag3/;
ENDIF;
ENDSTATE;

STATE diag3;
if #true;
call wait, #/sh_diags.avail/;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

STATE hndl_msg;
if #true;
DO_ACT clr_upd_integ;
call nilprim;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

ENDSELECT;
ENDLOOP
ENDSTATE;

410
VAR tsk_strt, integer;
VAR tsk_end, integer;
STAT cycle, integer;

ACTIVITY init;
  *time tsk_strt;
ENDACT;

ACTIVITY upd_cycle;
  *time tsk_end;
  sub tsk_end, tsk_strt, cycle;
  *time tsk_strt;
  move #0, num_chnls;
ENDACT;

ACTIVITY get_msg;
  delq waiting_q, msg_in;
  sizeq waiting_q, q.size;
  if (q.size = #0);
    move #0, msg_pending;
  endif;
ENDACT;

ACTIVITY queue_msg;
  addq msg_in, waiting_q, #1;
  move #1, msg_pending;
ENDACT;

ACTIVITY incr_chnl;
  add #1, num_chnls, num_chnls;
ENDACT;

ACTIVITY copy_msg;
  delq sh_diags.mages, msg_in;
ENDACT;

ACTIVITY chnl_req;
  move #11, msg_out.type;
  move #diag, msg_out.src;
  addq msg_out, sh_chnls.mages, #1;
ENDACT;

ACTIVITY choose_diag;
  rand #4, dg_num;
  move #21, msg_out.type;
  move #diag, msg_out.src;
  if (dg_num = #0);
    move #chnldiag, dest;
  else;
    if (dg_num = #1);
      move #chnldiag, dest;
    else;
      if (dg_num = #2);
        move #mgdiag, dest;
      else;
        move #tidiag, dest;
      endif;
    endif;
  endif;
ENDACT;

ACTIVITY send_diag;
  if (dg_num = #0);
    addq msg_out, sh_tonediag.mages, #1;
  else;
    if (dg_num = #1);
      addq msg_out, sh_chnldiag.mages, #1;
    else;
      if (dg_num = #2);
        addq msg_out, sh_mgiag.mages, #1;
      else;
        addq msg_out, sh_tidiag.mages, #1;
      endif;
    endif;
  endif;
ENDACT;

BEGIN
DO_ACT init;
  CALLP nilprim;
  SETSTATE #/clear/;
END;

STATE select:
  IF #true;
    DO_ACT upd_cycle;
    CALLP newline;
    SETSTATE #/start/;
  ENDIF;
ENDSTATE;

STATE clear:
  IF #true;
    CALLP get_msg;
    SETSTATE #/clear/;
  ENDIF;
ENDSTATE;

STATE start:
  IF (msg_pending = #1);
    DO_ACT get_msg;
    CALLP nilprim;
    SETSTATE #/get_msg/;
  ENDIF;
  IF #true;
    CALLP wait, #/sh_diags.avail/;
    SETSTATE #/resp1/;
  ENDIF;
ENDSTATE;

STATE resp1:
  IF #true;
    CALLP wait, #/sh_diags.mutex/;
    SETSTATE #/resp2/;
  ENDIF;
ENDSTATE;

STATE resp2:
  IF #true;
    DO_ACT copy_msg;
    CALLP signal, #/sh_diags.mutex/;
    SETSTATE #/get_msg/;
  ENDIF;
ENDSTATE;

STATE get_msg;
  IF #true;
    CALLP wait, #/sh_chnls.mutex/;
    SETSTATE #/get_m2/;
  ENDIF;
ENDSTATE;

STATE get_m2:
  IF #true;
    DO_ACT chnl_req;
    CALLP signal, #/sh_chnls.mutex/;
    SETSTATE #/get_m3/;
  ENDIF;
ENDSTATE;

STATE get_m3:
  IF #true;
    CALLP signal, #/sh_chnls.avail/;
    SETSTATE #/wt_ch1/;
  ENDIF;
ENDSTATE;

STATE wt_ch1:
  IF #true;
    CALLP wait, #/sh_diags.avail/;
    SETSTATE #/wt_ch2/;
  ENDIF;
ENDSTATE;

STATE wt_ch2:
  IF #true;
    CALLP wait, #/sh_diags.mutex/;
    SETSTATE #/wt_ch3/;
  ENDIF;
ENDSTATE;

STATE wt_ch3:
  IF #true;
    DO_ACT copy_msg;
    CALLP signal, #/sh_diags.mutex/;
    SETSTATE #/chk_msg/;
  ENDIF;
ENDSTATE;

STATE chk_msg,
IF (msg_in_type = $14);  
CALLP nilprim;
STATE #/ctnchg/a;
ENDIF;
IF $true;  
DO_ACT queue_msg;
CALLP nilprim;
STATE #/wt chiropr;
ENDIF;
ENDSTATE;

STATE ctnchg/a;
IF (num_chg/a = $1);  
CALLP nilprim;
STATE #/gct/chn/a;
ENDIF;
IF $true;  
DO_ACT incr_chn/a;
CALLP nilprim;
STATE #/gct/chn/a;
ENDIF;
ENDSTATE;

STATE get_chn/a;
IF $true;  
DO_ACT choose_diag;
CALLP nilprim;
STATE #/wt dia/a;
ENDIF;
ENDSTATE;

STATE wt dia/a;
IF (dg_num = $0);  
CALLP wait, #/sh_tenediag.mutex/a;
STATE #/pick dia/a;
ENDIF;
IF (dg_num = $1);  
CALLP wait, #/sh_tenediag.mutex/a;
STATE #/pick dia/a;
ENDIF;
IF (dg_num = $2);  
CALLP wait, #/sh_magdiag.mutex/a;
STATE #/pick dia/a;
ENDIF;
IF $true;  
CALLP wait, #/sh_tenediag.mutex/a;
STATE #/pick dia/a;
ENDIF;
ENDSTATE;

STATE pick dia/a;
IF (dg_num = $0);  
DO_ACT send_dia/a;
CALLP signal, #/sh_tenediag.mutex/a;
STATE #/pick2/a;
ENDIF;
IF (dg_num = $1);  
DO_ACT send_dia/a;
CALLP signal, #/sh_tenediag.mutex/a;
STATE #/pick2/a;
ENDIF;
IF (dg_num = $2);  
DO_ACT send_dia/a;
CALLP signal, #/sh_magdiag.mutex/a;
STATE #/pick2/a;
ENDIF;
IF $true;  
DO_ACT send_dia/a;
CALLP signal, #/sh_tenediag.mutex/a;
STATE #/pick2/a;
ENDIF;
ENDSTATE;

STATE pick2/a;
IF (dg_num = $0);  
CALLP signal, #/sh_tenediag.avail/a;
STATE #/clear/a;
ENDIF;
IF (dg_num = $1);  
CALLP signal, #/sh_tenediag.avail/a;
STATE #/clear/a;
ENDIF;
ENDSTATE;

ENDSTATE;
ENDSTATE;

STATE clear/a;
IF (dg_num = $2);  
CALLP signal, #/sh_magdiag.avail/a;
STATE #/clear/a;
ENDIF;
IF $true;  
CALLP signal, #/sh_tenediag.avail/a;
STATE #/clear/a;
ENDIF;
ENDSTATE;

ENDSTATE;

SYNCE Task

STAT rdy tm, integer;
STAT wt tm, integer;
STAT run tm, integer;
VAR msg out, msg alt;
VAR r msg, msg alt;
VAR res, string;
VAR tak strt, integer;
VAR tak end, integer;
STAT cycle, integer;

ENDSTATE;

ACTIVITY init;
ENDACT;
ACTIVITY upd cycle;
ENDACT;
ACTIVITY report sync;
ENDACT;
ACTIVITY clear sync;
ENDACT;
ACTIVITY change sync;
ENDACT;
ACTIVITY change sync;
ENDACT;
ACTIVITY set state;
ENDSTATE;

STATE start/a;
IF $true;  
CALLP sleep;
STATE #/slee/a;
ENDIF;
ENDSTATE;

STATE chk;
IF (sync loss = $1);  

412
BEGIN
  DD ACT init;
  CALLP nilprim;
  SETSTATE #/clear/;
END;
STATESELECT;

\| FAULT Task \\
STATE rdy_tm, integer;
STATE wt_tm, integer;
STATE run_tm, integer;
VAR msg_out, t_msg_splt;
VAR msg_in, t_msg_splt;
VAR r_msg, t_msg_splt;
VAR recv, string;
VAR takstrt, integer;
VAR takend, integer;
STATE cycle, integer;

ACTIVITY init:
  +time takstrt;
ENDACT;

ACTIVITY upd_cycle:
  +time tak_end;
  +sub tak_end, takstrt, cycle;
  +time takstrt;
ENDACT;

ACTIVITY copy_in:
  delq sh_fault;msges, msg_in;
ENDACT;

ACTIVITY send_report:
  move #7, msg.out.type;
  move #/faulty, msg.out.src;
  addq msg.out, sh_msg, msges, $1;
ENDACT;
BEGIN
  DD ACT init;
END;
STATE clear;
STATE start;
STATE respond;
STATE resp1;
STATE resp2;
STATE resp3;
ENDSTATE;

\| AUDITS Task \\
STATE rdy_tm, integer;
STATE wt_tm, integer;
STATE run_tm, integer;

413
VAR msg_out, t_msg;  // msg_out, t_msg
VAR r_msg, t_msg;    // r_msg, t_msg
VAR rcv, string;    // rcv, string
VAR aud1_cnt, integer; // aud1_cnt, integer
VAR aud2_cnt, integer; // aud2_cnt, integer
VAR aud3_cnt, integer; // aud3_cnt, integer
VAR task_strt, integer; // task_strt, integer
VAR task_end, integer; // task_end, integer
STAT cycle, integer;

ACTIVITY init;  // time task_strt
ENDT;
ACTIVITY upd_cycle;  // time task_end
*sub task_end, task_strt, cycle;
*time task_strt;
ENDT;
ACTIVITY incr_cnt;  // add $1, aud1_cnt, aud1_cnt;
add $1, aud2_cnt, aud2_cnt;
add $1, aud3_cnt, aud3_cnt;
ENDT;
ACTIVITY tell_tmto;  // move &aud1, msg_out.type;
move &aud2, msg_out.type;
move &aud3, msg_out.type;
addq msg_out, sh_tmto.mages, $1;
ENDT;
ACTIVITY aud1;  // move &0, aud1_cnt;
move &0, msg_out.type;
move &aud1, msg_out.type;
addq msg_out, sh_diags.mages, $1;
ENDT;
ACTIVITY aud2;  // move &0, aud2_cnt;
move &0, msg_out.type;
move &aud2, msg_out.type;
addq msg_out, sh_diags.mages, $1;
ENDT;
ACTIVITY aud3;  // move &0, aud3_cnt;
move &0, msg_out.type;
move &aud3, msg_out.type;
addq msg_out, sh_diags.mages, $1;
ENDT;
BEGIN
DO.ACT init;
CALLP nilprim;
SETSTATE #/clear/;
END:
STATESELECT;
STATE clear;  // upd_cycle;
DO.ACT newline;
SETSTATE #/start/;
ENDIF;
ENDSTATE;
STATE start;  // incr_cnt;
DO.ACT newline;
CALLP nilprim;
SETSTATE #/trigger_time/;
ENDIF;
ENDSTATE;
STATE trigger_time;  // IF $true;
DO.ACT newline;
CALLP null, #/sh_tmto.mates/;
SETSTATE #/tx2/;
ENDIF;
ENDSTATE;
STATE tx2;  // IF $true;
DO.ACT tell_tmto;
CALLP null, #/sh_tmto.mutes/;
SETSTATE #/tx3/;
ENDIF;
ENDSTATE;
STATE tx3;  // IF $true;
CALLP null, #/sh_tmto.avail/;
SETSTATE #/chk audits/;
ENDIF;
ENDSTATE;
STATE chk audits;  // IF (aud1_cnt == 1)
CALLP null, #/sh_diags.mutes/;
SETSTATE #/chk audits/;
ENDIF;
ENDSTATE;
STATE chk audits2;  // IF $true;
DO.ACT aud1;
CALLP null, #/sh_diags.mutes/;
SETSTATE #/chk audits2/;
ENDIF;
ENDSTATE;
STATE chk audits3;  // IF $true;
CALLP null, #/sh_diags.avail/;
SETSTATE #/chk audits2/;
ENDIF;
ENDSTATE;
STATE chk audits4;  // IF (aud2_cnt == 20)
CALLP null, #/sh_diags.mutes/;
SETSTATE #/chk audits/;
ENDIF;
ENDSTATE;
STATE chk audits22;  // IF $true;
DO.ACT aud3;
CALLP null, #/sh_diags.mutes/;
SETSTATE #/chk audits22/;
ENDIF;
ENDSTATE;
STATE chk audits23;  // IF $true;
CALLP null, #/sh_diags.avail/;
SETSTATE #/chk audits22/;
ENDIF;
ENDSTATE;
STATE chk audits33;  // IF (aud3_cnt == 30)
CALLP null, #/sh_diags.mutes/;
SETSTATE #/chk audits23/;
ENDIF;
ENDSTATE;
STATE chk audits3;  // IF $true;
DO.ACT aud3;
CALLP null, #/sh_diags.mutes/;
SETSTATE #/chk audits3/;
ENDIF;
ENDSTATE;
STATE chk audits2;  // IF $true;
DO.ACT aud2;
CALLP null, #/sh_diags.mutes/;
SETSTATE #/chk audits2/;
ENDIF;
ENDSTATE;
STATE chk audits1;  // IF $true;
DO.ACT aud1;
CALLP null, #/sh_diags.mutes/;
SETSTATE #/chk audits1/;
ENDIF;
ENDSTATE;
STATE chk audits;  // IF $true;
DO.ACT aud1;
CALLP null, #/sh_diags.mutes/;
SETSTATE #/chk audits/;
ENDIF;
ENDSTATE;
STATE chk_33;
  IF #true;
    CALLP signal, #/sh_diags.avail/;
    SETSTATE #/clear/;
  ENDIF;
ENDSTATE;
ENDLOOP
ENDTASK;

; ; TINTC Task
; ;
STAT rdy_tm, integer;
STAT ut_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_elts;
VAR msg_in, t_msg_elts;
VAR r_msg, t_msg_elts;
VAR recv, string;
VAR result, integer;
VAR fail, integer;
VAR task_start, integer;
VAR task_end, integer;
STAT cycle, integer;

ACTIVITY init;
  *time task_start;
ENDACT;

ACTIVITY upd_cycle;
  *time task_end;
  *sub task_end, task_start, cycle;
  *time task_start;
ENDACT;

ACTIVITY copy_in;
  delq sh_tintc.msgs, msg_in;
ENDACT;

ACTIVITY do_scan;
  move 00, fail;
  move $1, ti_data;
  ticks $100;
  rand $10, result;
  if (result = $9);
    move $1, fail;
  endif;
ENDACT;

ACTIVITY report_fail;
  move $10, msg_out.type;
  move #/tintc/, msg_out.src;
  addq msg_out, sh_fault.mags, $1;
ENDACT;

BEGIN
  DU.ACT init;
  CALLP nilprim;
  SETSTATE #/clear/;
ENDLOOP
ENDTASK;

; ; TDIAG Task
; ;
STATE clear;
  IF #true;
    DU.ACT upd_cycle;
    SETSTATE #/start/;
  ENDIF;
ENDSTATE;

STATE start;
  IF #true;
    CALLP wait, #/sh_tintc.avail/;
    SETSTATE #/respond/;
  ENDIF;
ENDSTATE;

STATE respond;
  IF #true;
    CALLP copy_in;
    SETSTATE #/got_msg/;
  ENDIF;
ENDSTATE;

STATE got_msg;
  IF (msg_in.type = $23);
    DU.ACT do_asnm;
    CALLP nilprim;
    SETSTATE #/chk_results/;
  ENDIF;
  IF #true;
    CALLP nilprim;
    SETSTATE #/clear/;
  ENDIF;
ENDSTATE;

STATE chk_results;
  IF (fail = $1);
    CALLP wait, #/sh_faults.avail/;
    SETSTATE #/chk2/;
  ENDIF;
  IF #true;
    CALLP nilprim;
    SETSTATE #/respond/;
  ENDIF;
ENDSTATE;

STATE chk2;
  IF #true;
    DU.ACT report_fail;
    CALLP signal, #/sh_faults.avail/;
    SETSTATE #/chk3/;
  ENDIF;
ENDSTATE;

STATE chk3;
  IF #true;
    CALLP signal, #/sh_fault.avail/;
    SETSTATE #/respond/;
  ENDIF;
ENDSTATE;

STATE respond;
  IF (fail = $0);
    CALLP endresp, #7;
    SETSTATE #/clear/;
  ENDIF;
  IF #true;
    CALLP nilprim;
    SETSTATE #/clear/;
  ENDIF;
ENDSTATE;
DO_ACT stop_mtc;
CALLP signal, #/sh_signal.mutex/;
SETSTATE #/got2/;
ENDIF;
ENDSTATE;

STATE got3;
IF #true;
CALLP signal, #/sh_signal.avail/;
SETSTATE #/diagnose/;
ENDIF;
ENDSTATE;

STATE diagnose;
IF #true;
DO_ACT do_diag;
CALLP nilprim;
SETSTATE #/chk_results/;
ENDIF;
ENDSTATE;

STATE chk_results;
IF (fail = #1);
CALLP wait, #/sh_fault.mutex/;
SETSTATE #/chk2/;
ENDIF;
IF #true;
CALLP nilprim;
SETSTATE #/response/;
ENDIF;
ENDSTATE;

STATE chk2;
IF #true;
DO_ACT report_fail;
CALLP signal, #/sh_fault.mutex/;
SETSTATE #/chk3/;
ENDIF;
ENDSTATE;

STATE chk3;
IF #true;
CALLP signal, #/sh_fault.avail/;
SETSTATE #/response/;
ENDIF;
ENDSTATE;

STATE response;
IF (fail = #0);
CALLP endresp, #0;
SETSTATE #/clear/;
ENDIF;
IF #true;
CALLP nilprim;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

ENDSELECT;
ENDLOOP
ENDTASK;

MSGDIAG Task

STAT rd_tm, integer;
STAT st_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_elt;
VAR msg_in, t_msg_elt;
VAR x_msg, t_msg_elt;
VAR rer, string;
VAR result, integer;
VAR fail, integer;
VAR tak_strt, integer;
VAR tak_end, integer;
STAT cycle, integer;

ACTIVITY init;
+time  tak_strt;
ENDACT;

ACTIVITY upd_cycle;
+time  tak_end;
+sub  tak_end, tak_strt, cycle;
+time  tak_strt;
ENDACT;

ACTIVITY stop_integ;
move $20, mag_out.type;
move #/magdiag/, mag_out.src;
addq mag_out, sh_signal.mages, #1;
ENDACT;

ACTIVITY copy_in;
deln sh_magdiag.mages, mag_in;
ENDACT;

ACTIVITY do_diag;
move $0, fail;
ticks $250;
rand $10, result;
if (result = $0);
move #1, fail;
endif;
ENDACT;

ACTIVITY report_fail;
move #10, mag_out.type;
move #/magdiag/, mag_out.src;
addq mag_out, sh_fault.mages, #1;
ENDACT;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
LOOP;
STATESELECT;
STATE clear;
if #true;
DO_ACT upd_cycle;
CALLP niwp;
SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
if #true;
DO_ACT wait, #/sh_magdiag_avail/;
SETSTATE #/respond/;
ENDIF;
ENDSTATE;

STATE respond;
if #true;
DO_ACT wait, #/sh_magdiag_avail/;
SETSTATE #/res2/;
ENDIF;
ENDSTATE;

STATE resp2;
if #true;
DO_ACT copy_in;
CALLP signal, #/sh_magdiag.mutes/;
SETSTATE #/got_msg/;
ENDIF;
ENDSTATE;

STATE got_msg;
if #true;
DO_ACT wait, #/sh_signal.mutes/;
SETSTATE #/got2/;
ENDIF;
ENDSTATE;

STATE got2;
if #true;
DO_ACT stop_integ;
CALLP signal, #/sh_signal.mutes/;
SETSTATE #/got3/;
ENDIF;
ENDSTATE;

ENDSTATE;

STATE got3;
if #true;
CALLP signal, #/sh_signal_avail/;
SETSTATE #/diagnose/;
ENDIF;
ENDSTATE;

STATE diagnose;
if #true;
DO_ACT do_diag;
CALLP nilprim;
SETSTATE #/chk_results/;
ENDIF;
ENDSTATE;

STATE chk_results;
if (fail = $1);
CALLP wait, #/sh_fault.mutes/;
SETSTATE #/chk2/;
ENDIF;
if #true;
CALLP nilprim;
SETSTATE #/resp/;
ENDIF;
ENDSTATE;

STATE chk2;
if #true;
DO_ACT report_fail;
CALLP signal, #/sh_fault.mutes/;
SETSTATE #/chk3/;
ENDIF;
ENDSTATE;

STATE chk3;
if #true;
CALLP signal, #/sh_fault.mutes/;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

STATE resp;
if (fail = $0);
CALLP endresp, #0;
SETSTATE #/clear/;
ENDIF;
if #true;
CALLP nilprim;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

ENDSTATE;

ENDSTATE;

ENDSTATE;

ENDSTATE;

ENDSTATE;

ENDSTATE;

STATE start;
if #true;
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

STATE end;
if #true;
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

ENDSTATE;

ENDSTATE;

ENDSTATE;

SEND;

ENDSTATE;

ENDSTATE;

ENDSTATE;

ENDSTATE;

ENDSTATE;

ENDSTATE;

ENDSTATE;

ENDSTATE;

SEND;

ENDSTATE;

ENDSTATE;

SEND;

ENDSTATE;

ENDSTATE;

ENDSTATE;

SEND;

ENDSTATE;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;

ENDSTATE;

SEND;
*time tsk_strt;
ENDACT;

ACTIVITY copy_in;
delq sh_tonediagmsgs, msg_in;
END;

ACTIVITY select_tone;
move #2, msg_out.type;
mov #/tonedlg/, msg_out.src;
addd msg_out, sh_tonesmsgs, #1;
END;

ACTIVITY do_diag;
mov #0, fail;
ticks #300;
verb #10, result;
if (result = #0);
mov #1, fail;
endif;
END;

ACTIVITY report_fail;
mov #10, msg_out.type;
mov #/tonedlg/, msg_out.src;
addd msg_out, sh_faultmsgs, #1;
END;

BEGIN
DO_ACT init;
callp nilprim;
SETSTATE #/clear/;

LOOP;
STATESELECT;

STATE clear;
if #true;
DO_ACT
updt_cycle;
callp newresp;
SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
if #true;
callp wait, #/sh_tonediag.avail/;
SETSTATE #/respond/;
ENDIF;
ENDSTATE;

STATE respond;
if #true;
callp wait, #/sh_tonediag.avail/;
SETSTATE #/respl/;
ENDIF;
ENDSTATE;

STATE resp1;
if #true;
DO_ACT copy_in;
callp signal, #/sh_tonediag.avail/;
SETSTATE #/getmsg/;
ENDIF;
ENDSTATE;

STATE gotmsg;
if #true;
callp wait, #/sh_tonesmsgs/;
SETSTATE #/get2/;
ENDIF;
ENDSTATE;

STATE got2;
if #true;
DO_ACT select_tone;
callp signal, #/sh_tonesmsgs/;
SETSTATE #/got3/;
ENDIF;
ENDSTATE;

STATE got3;
if #true;
callp signal, #/sh_tonesmsgs/;
SETSTATE #/diagnosis/;

ENDIF;
ENDSTATE;

STATE diagnose;
if #true;
d定居 do_diag;
callp nilprim;
SETSTATE #/chk_resuts/;
ENDIF;
ENDSTATE;

STATE chk_result;
if (fail = #1);
callp wait, #/sh_faultmsgs/;
SETSTATE #/chk2/;
ENDIF;
if #true;
callp nilprim;
SETSTATE #/response/;
ENDIF;
ENDSTATE;

STATE chk2;
if #true;
d定居 report_fail;
callp signal, #/sh_faultmsgs/;
SETSTATE #/chk3/;
ENDIF;
ENDSTATE;

STATE chk3;
if #true;
callp signal, #/sh_faultmsgs/;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

STATE response;
if (fail = #0);
callp endresp, #6;
SETSTATE #/clear/;
ENDIF;
if #true;
callp nilprim;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

ENDSTATE;

ENDDO
ENDTASK;

| CENLDIAG Task | | STAT rdv_tm, integer;
STAT wv_tm, integer;
STAT run_tm, integer;
VAR msg_out, tmsg_alt;
VAR msg_in, tmsg_alt;
VAR tmsg, tmsg_alt;
VAR res, string;
VAR result, integer;
VAR fail, integer;
VAR tsk_strt, integer;
VAR tsk_end, integer;
STAT cycle, integer;

ACTIVITY init;
*time tsk_strt;
ENDACT;

ACTIVITY updt_cycle;
*time tsk_strt;
sub tsk_end, tsk_strt, cycle;
cicle tsk_strt;
ENDACT;

ACTIVITY copy_in;
delq sh_cenldiagmsgs, msg_in;

418
ENDACT;

ACTIVITY connect_chnl;
move $11, msg_out.type;
move #/chndldiag, msg_out.src.;
addq msg_out, sh_connect.msges, $1;
ENDACT;

ACTIVITY stop_integ;
move $20, msg_out.type;
move #/chndldiag, msg_out.src.;
addq msg_out, sh_signal.msges, $1;
ENDACT;

ACTIVITY do_diag;
move $0, fail;
ticks $150;
rnd $10, result;
if (result = $9); move $1, fail;
endif;
ENDACT;

ACTIVITY report_fail;
move $10, msg_out.type;
move #/chndldiag, msg_out.src.;
addq msg_out, sh_fault.msges, $1;
ENDACT;

BEGIN DO_ACT int;
callp nilprim;
SETSTATE #/clear/;
ENDACT;

LOOP; STATESSELECT;

STATE clear;
if (true);
doa upd_cycle;
callp noresp;
SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
if (true);
callp wait, #/sh_chndl.diag.avail/;
SETSTATE #/resp/;
ENDIF;
ENDSTATE;

STATE respond;
if (true);
callp wait, #/sh_chndl.diag.mutex/;
SETSTATE #/rep/;
ENDIF;
ENDSTATE;

STATE rep/;
if (true);
doa copy_in;
callp signal, #/sh_chndl.diag.mutex/;
SETSTATE #/got_msg/;
ENDIF;
ENDSTATE;

STATE got_msg;
if (true);
callp wait, #/sh_connect.mutex/;
SETSTATE #/got/0/;
ENDIF;
ENDSTATE;

STATE got/0;
if (true);
doa connect_chnl;
callp signal, #/sh_connect.mutex/;
SETSTATE #/got/0/;
ENDIF;
ENDSTATE;

STATE got/0;
if (true);
callp signal, #/sh_connect.avail/;
SETSTATE #/sig/0/;
ENDIF;
ENDSTATE;

STATE sig;
if (true);
callp wait, #/sh_signal.mutex/;
SETSTATE #/sig/0/;
ENDIF;
ENDSTATE;

STATE sig/0;
if (true);
callp signal, #/sh_signal.mux/;
SETSTATE #/sig/0/;
ENDIF;
ENDSTATE;

STATE diag;
if (true);
callp signal, #/sh_diag.avail/;
SETSTATE #/diag/0/;
ENDIF;
ENDSTATE;

STATE diag/0;
if (true);
callp signal, #/sh_diag.mux/;
SETSTATE #/diag/0/;
ENDIF;
ENDSTATE;

STATE chk_results;
if (fail = $1);
callp wait, #/sh_fault.mux/;
SETSTATE #/chk_result/0/;
ENDIF;
ENDSTATE;

STATE chk_result/0;
if (true);
callp report_fail;
callp nilprim;
SETSTATE #/chk_result/0/;
ENDIF;
ENDSTATE;

STATE chk/2;
if (true);
callp signal, #/sh_fault.mux/;
SETSTATE #/chk2/0/;
ENDIF;
ENDSTATE;

STATE chk/2;
if (true);
callp signal, #/sh_fault.mux/;
SETSTATE #/chk2/0/;
ENDIF;
ENDSTATE;

STATE resp;
if (fail = $0);
callp endresp, $0;
SETSTATE #/clear/0/;
ENDIF;
ENDSTATE;

STATE resp/0;
if (true);
callp nilprim;
SETSTATE #/clear/0/;
ENDIF;
ENDSTATE;

STATE got2;
if (true);
callp signal, #/sh_connect.mux/;
SETSTATE #/got/3/0/;
ENDIF;
ENDSTATE;

STATE got/3;
if (true);
callp signal, #/sh_connect.avail/;
SETSTATE #/sig/3/0/;
ENDIF;
ENDSTATE;
B.4 Scheduling Analysis Software

This section contains all of the Design-Aid code used to model the simulations presented in the scheduling analysis part of the design methodology chapter.

B.4.1 Baseline System Software

The kernel used to obtain the baseline scheduling results is that presented in section B.2.2. The application software used to obtain the baseline scheduling results is that presented in section B.2.3.

B.4.2 Timeslicing Kernel

The kernel used to implement time sliced scheduling is exactly the same as that used in section B.2.2 with the addition of the following interrupt handling primitive.

```
PRIM evt_slc;
  disable #10;
  askint #5;
  +add #1, evt_5, evt_5;
  +time start_os;
  +time curr_tcb sts.finish;
  +sub curr_tcb sts.finish, curr_tcb sts.begin, curr_tcb sts.begin;
  +add curr_tcb sts.begin, curr_tcb sts.run_time, curr_tcb sts.run_time;
  +move curr_tcb sts.run_time, G run_str;
  +time curr_tcb sts.begin;
  pop taskstk, ret_pc;
  move taskstk, curr_tcb stk;
  move ret_pc, curr_tcb.pcnt;
  call sched_exec_task, curr_tcb;
  call run_tcb_task;
ENDPRIM;
bind evt_slc, #5;
```

B.4.3 Response Priority Kernel

The following node file contains the global type definitions used in the response priority kernel.

```
; NODE FILE
;
; t_trace_elt.act Values:
; 1 - Mag Sent
; 2 - Mag Accepted
; 3 - Shared Data Write
; 4 - Shared Data Read
; 5 - Response Unit
; 6 - Event Trigger
; 7 - Start Read Delimiter
; 8 - End Read Delimiter
; 9 - Start Mag Delimiter
; 10 - End Mag Delimiter

TYPE t_trace_elt = node_name:string *;
  task_name:string *;
  act :integer *;
  time :integer *;
  mix1 :integer *;
  mix2 :integer *;

TYPE t_mag_elt = data : integer *;
  type : integer *;
  prior : integer *;
  arc : string *;
  trc_size : integer; No. of associated trace elts
```

420
The following kernel was used to implement scheduling based on response priorities instead of individual task priorities. The response priorities are propagated by message tags in the msg.prior field.

KERNEL CODE FILE

This kernel contains the following primitives:
- nonblocking send
- blocking selective receive
- sleep
- get data from another task
- nilprim

The following external message types are supported:
- #1 - start call setup
- #2 - provide tone
- #3 - run diagnostic
- #4 - integrity ok
- #5 - integrity failure
- #6 - call takedown

The following internal message types are supported:
- #1 - START - request call setup
- #2 - TONE - request for tone
- #3 - DIAG - request single random diagnostic
- #4 -
- #5 -
- #6 -
- #7 - MSG - report fault
- #8 - MSG - perform and report synch operation
- #9 - FAULT - synch loss
- #10 - FAULT - general fault report
- #11 - CONNECT - perform connect/disconnect
- #12 - CHELS - allocate channel
- #13 - CHELS - free channel
- #14 - all tasks - channel granted
- #15 - STAT - call setup complete
- #16 - SIGNAL - start integrity checking
- #17 - STAT - integrity failure report
- #18 - END - caller hung up
- #19 - STAT - call finished
- #20 - STAT - stop integrity checking
- #21 - all diag - run diagnostic
- #22 - TINTC - stop carrier checking
- #23 - TINTC - monitor carriers

The following response types are supported:
- #1 - MSG - fault report sent to switching center
- #2 - MSG - synch source changed
- #3 - CONNECT - switching center tone request connected
- #4 - STAT - call finished
- #5 - STAT - call started
- #6 - alldiag - diagnostic completed without failure
- #7 - TINTC - carrier scan completed without failure
TYPE t_tcb =
  rdy_time : integer *;
  run_time : integer *;
  wt_time : integer *;
  begin : integer *;
  finish : integer *;
  resp_q : queue ;
  of t_trace_elt;

TYPE t_shared =
  data_trace : queue ;
  of t_trace_elt;

TYPE t_tcb =
  num : string *
  num : integer *;
  status : string *
  status : integer *;
  priority : integer *
  priority :
  pcnt : integer *
  program_counter :
  elt_str : string *
  rev msg elt :
  stk : stack *
  rev msg elt :
  stack :
  sta : t_tcb
  of t_tcb_stats :
  task control block;

TYPE t_sh_data =
  task_num : string *
  task_num :
  num_msg : integer *
  num msg :
  msg : queue ;
  queue :
  trace : queue ;
  trace queue (t_trace_elt);

TYPE t_msg_struct =
  task_num : string *
  task_num :
  waiting : integer *
  msg_q : queue *
  flag set if blocked for msg :
  msg_queue (t_msg_elt) :
  trace : queue (t_trace_elt) :
  trace_queue (t_trace_elt) :
  type : integer *
  type of msg blocked on :
  of t_tcb :
  tcb wait area;

TYPE t_id_struct =
  num : string *
  of t_id_struct (one elt per task);
  trc_name : string *
  st_name : string ;

SHARED comm_q, queue ;
SHARED sh_{21}, t_sh_data;
SHARED sh_{22}, t_sh_data;
SHARED sh_{23}, t_sh_data;
SHARED sh_{24}, t_sh_data;
SHARED sh_{25}, t_sh_data;
SHARED sh_{26}, t_sh_data;
SHARED sh_{27}, t_sh_data;
SHARED sh_{28}, t_sh_data;
SHARED sh_{29}, t_sh_data;
SHARED sh_{30}, t_sh_data;
SHARED sh_{31}, t_sh_data;
SHARED sh_{32}, t_sh_data;
SHARED sh_{33}, t_sh_data;
SHARED sh_{34}, t_sh_data;
SHARED sh_{35}, t_sh_data;
SHARED sh_{36}, t_sh_data;
SHARED sh_{37}, t_sh_data;
SHARED sh_{38}, t_sh_data;
SHARED sh_{39}, t_sh_data;
SHARED sh_{40}, t_sh_data;
SHARED sh_{41}, t_sh_data;
SHARED sh_{42}, t_sh_data;
SHARED sh_{43}, t_sh_data;
SHARED sh_{44}, t_sh_data;
SHARED sh_{45}, t_sh_data;
SHARED sh_{46}, t_sh_data;
SHARED sh_{47}, t_sh_data;
SHARED sh_{48}, t_sh_data;
SHARED sh_{49}, t_sh_data;
SHARED sh_{50}, t_sh_data;
SHARED sh_{51}, t_sh_data;
SHARED sh_{52}, t_sh_data;
SHARED sh_{53}, t_sh_data;
SHARED sh_{54}, t_sh_data;
SHARED sh_{55}, t_sh_data;
SHARED sh_{56}, t_sh_data;
SHARED sh_{57}, t_sh_data;
SHARED sh_{58}, t_sh_data;
SHARED sh_{59}, t_sh_data;
SHARED sh_{60}, t_sh_data;
SHARED sh_{61}, t_sh_data;
SHARED sh_{62}, t_sh_data;
SHARED sh_{63}, t_sh_data;
SHARED sh_{64}, t_sh_data;
SHARED sh_{65}, t_sh_data;
SHARED sh_{66}, t_sh_data;
SHARED sh_{67}, t_sh_data;

; Message queues
VAR loc_comm_q, queue ;
VAR st_elt, t_id_struct ;
VAR l_{?1}, t_tcb ;
VAR l_{?2}, t_msg_struct ;
VAR l_{?3}, t_msg_struct ;
VAR l_{?4}, t_msg_struct ;
VAR l_{?5}, t_msg_struct ;
VAR l_{?6}, t_msg_struct ;
VAR l_{?7}, t_msg_struct ;
VAR l_{?8}, t_msg_struct ;
VAR l_{?9}, t_msg_struct ;
VAR l_{?10}, t_msg_struct ;
VAR l_{?11}, t_msg_struct ;
VAR l_{?12}, t_msg_struct ;
VAR l_{?13}, t_msg_struct ;
VAR l_{?14}, t_msg_struct ;
VAR l_{?15}, t_msg_struct ;
VAR l_{?16}, t_msg_struct ;
VAR local, string;
VAR shared, string;
VAR loc_trc, string;
VAR sh_trc, string;

STAT sends, integer;
STAT os_ovhd, integer;
STAT sc_ovhd, integer;
STAT switches, integer;
STAT preempts, integer;
STAT evt_0, integer;
STAT evt_1, integer;
STAT evt_2, integer;
STAT evt_3, integer;
STAT evt_4, integer;
STAT evt_5, integer;
VAR start_os, integer;
VAR end_os, integer;
VAR elapsed, integer;
VAR sc_start, integer;
VAR sc_end, integer;
VAR rdy_str, string;
VAR wt_str, string;
VAR run_str, string;
VAR end_str, string;
VAR wq, queue;
VAR wt_q, queue;
VAR msg_str, string;
VAR dest_str, string;
VAR temp_tcb, t_tcb;
VAR temp2_tcb, t_tcb;
VAR res_pc, integer;
VAR tmp stk, stack;
VAR num_parms, integer;
VAR found, integer;
VAR msg_type, integer;
VAR data_size, integer;
VAR nil_queue, queue;
VAR nil_string, string;
VAR msg_elt, t_msg_elt;
VAR m_num, integer;
VAR temp2_q, queue;
VAR rtype, integer;
VAR q_size, integer;
VAR done, integer;

: Global Kernel Variables
VAR cure_tcb, t_tcb ; current task control block
VAR proclist, queue ; copy of task name list
VAR br_q, queue ; branch condition check
VAR tempstr, string ; temporary string value
VAR tcb_name, string ; current task name
VAR rtx, queue ; ready to run queue

: Variables for get_local and get_sh procedures
VAR g_q_size, integer
VAR g_found, integer
VAR g_msg_elt, t_id_struct

: Variables for response_update procedure
VAR action, integer ; action requested in response_update
VAR prio, integer ; send priority
VAR trace_elt, t_trace_elt ; new response trace element
VAR msg_elt_name, string ; name of msg elt for response trace
VAR dest_elt_name, string ; name of msg queue
VAR delimiter, t_trace_elt ; msg or data trace delimiter
VAR q_size, integer ; size of trace queue
VAR data_name, string ; name of shared data variable
VAR temp_q, queue ; data response queue copy

: VARIABLE declarations for procedure RUN_RTR_TASK
VAR error, boolean ; Set if no ready processes

: VARIABLE declarations for procedure SCHED_RTR_TASK
VAR rtx_tcb, t_tcb ; PCB to be scheduled

: PROCEDURE response_update (name : string;
; action: integer)
This procedure updates the response trace queues

PROC response_update;
  *pop taskstk, action;
  *move nod_name, trace_elt.node_name;
  *move curr_tcb.name, trace_elt.task_name;
  *move trace_elt.time, trace_elt.time;
  *move action, trace_elt.act;

  *if (action = $1); SEND MSG
    *pop taskstk, prio;
    *pop taskstk, dest_elt_name;
    *pop taskstk, msg_elt_name;
    *addq trace_elt, curr_tcb.ste.resp_q, #1;
    *move curr_tcb.ste.resp_q, temp_q;
    *sizeq temp_q, q.size;
    *move q.size, msg_elt_name.trc.size;
    *while (q.size > 0);
      *delq temp_q, trace_elt;
      *addq trace_elt, @dest_elt_name, prio;
      *sizeq temp_q, q.size;
    endwhile;
  endif;

  *if (action = $2); RCV MSG
    *pop taskstk, dest_elt_name;
    *pop taskstk, msg_elt_name;
    *move trace_elt, delim_elt;
    *addq trace_elt, curr_tcb.ste.resp_q, #1;
    *move $9, delim_elt.act;
    *addq delim_elt, curr_tcb.ste.resp_q, #1;
    *move msg_elt_name.trc.size, q.size;
    *while (q.size > 0);
      *delq @dest_elt_name, trace_elt;
      *addq trace_elt, curr_tcb.ste.resp_q, #1;
      *subb q.size, #1, q.size;
    endwhile;
    *move $10, delim_elt.act;
    *addq delim_elt, curr_tcb.ste.resp_q, #1;
  endif;

  *if (action = $3); DATA WRITE
    *pop taskstk, data_name;
    *addq trace_elt, curr_tcb.ste.resp_q, #1;
    *move curr_tcb.ste.resp_q, @data_name.stats.trace_q;
  endif;

  *if (action = $4); DATA READ
    *pop taskstk, data_name;
    *move trace_elt, delim_elt;
    *addq trace_elt, curr_tcb.ste.resp_q, #1;
    *move $7, delim_elt.act;
    *addq delim_elt, curr_tcb.ste.resp_q, #1;
    *move @data_name.stats.data_trace, temp_q;
    *sizeq temp_q, q.size;
    *while (q.size > 0);
      *delq temp_q, trace_elt;
      *addq trace_elt, curr_tcb.ste.resp_q, #1;
      *sizeq temp_q, q.size;
    endwhile;
    *move $6, delim_elt.act;
    *addq delim_elt, curr_tcb.ste.resp_q, #1;
  endif;

  *if (action = $5); RESPONSE OUT
    *pop taskstk, trace_elt.misc1;
    *addq trace_elt, curr_tcb.ste.resp_q, #1;
    *sizeq curr_tcb.ste.resp_q, q.size;
    *while (q.size > 0);
      *delq curr_tcb.ste.resp_q, trace_elt;
    *type $/task: /;
    *type trace_elt.task_name;
    *type $/ action: /;
    *type trace_elt.act;
    *type $/ time: /;
    *type trace_elt.time;
    *type $/ misc1: /;
    *type trace_elt.misc1;
    *type $/ misc2: /;

  endif;

  *endproc;

424
; PROCEDURE run_rtr_task
; This procedure takes the first ready to run task off the rtr queue (this is the one with the highest priority) and copies its saved state into the current processor state. If there are no ready to run tasks, then an error message is printed and the kernel terminates.
PROC run_rtr_task;
  @time mc_start;
  @sizeq rtr, br_chk;
  IF (br_chk = 80);
    printer $/ERROR: NO RTR PROCESSES/;
    halt;
  ELSE;
    delq rtr, curr_tcb;
    move curr_tcb.name, tcb_name;
    outw tcb_name;
    add w1, switches, switches;
    @time curr_tcb.sts.finish;
    @sub curr_tcb.sts.finish, curr_tcb.sts.begin, curr_tcb.sts.begin;
    @add curr_tcb.sts.begin, curr_tcb.sts.rdy_time, curr_tcb.sts.rdy_time;
    @move $/rdy tm, rdy_str;
    @move curr_tcb.sts.rdy_time, @rdy_str;
    @move curr_tcb.sts.st_time, @st_str;
    @time curr_tcb.sts.begin;
    move $/running/, curr_tcb.status;
    move curr_tcb.stk, taskstk;
    @time end_cs;
    @sub end.cs, start.cs, elapsed;
    add elapsed, os_ovhd, os_ovhd;
    @time os_end;
    @sub os_end, sc_start, sc_start;
    @add sc_start, sc_ovhd, sc_ovhd;
    @enable $10;
    move curr_tcb.pcnt, pc;
  ENDIF;
ENDPROC;

; PROCEDURE sched_rtr_task (rtr_tcb : t_tcb)
; This procedure accepts a process control block and adds it to the ready to run queue in a prioritized fashion.
PROC sched_rtr_task;
  @time sc_start;
  pop taskstk, rtr_tcb;
  move $/ready/, rtr_tcb.status;
  @time rtr_tcb.sts.finish;
  @sub rtr_tcb.sts.finish, rtr_tcb.sts.begin, rtr_tcb.sts.begin;
  @add rtr_tcb.sts.begin, rtr_tcb.sts.st_time, rtr_tcb.sts.st_time;
  move taskstk, temp stk;
  @time rtr_tcb.sts.begin;
  addq rtr_tcb, rtr_tcb.priority;
  IF (rtr_tcb.priority > curr_tcb.priority);
    pop taskstk, ret_pc;
    pop taskstk, ret_pc;
    move ret_pc, curr_tcb.pcnt;
    move taskstk, curr_tcb.stk;
  @time curr_tcb.sts.finish;
  @sub curr_tcb.sts.finish, curr_tcb.sts.begin, curr_tcb.sts.begin;
  @add curr_tcb.sts.begin, curr_tcb.sts.run_time, curr_tcb.sts.run_time;
  move curr_tcb.sts.run_time, @run_str;
  @time curr_tcb.sts.begin;

425
move #/ready/, curr_tcb.status;
addq curr_tcb, rtr, curr_tcb.priority;
*add #1, preemptx, preemptx;
*sub sc_end, sc_start, sc_start;
*add sc_start, sc_ovhd, sc_ovhd;
call runstrtcb;
halt;
endif;
*sub sc_end, sc_start, sc_start;
*add sc_start, sc_ovhd, sc_ovhd;

; PROCEDURE get_sh (taskname : string)
; This procedure accepts a task name and sets the shared and
; sh_trc variables to hold the names of the task's messaging and trace
; structures.
PROC get_sh;
pop taskstk, tempstr;
move #0, g_found;
lock;
siseq comm_q, g_q_size;
while (g_q_size > #0) & ("g_found");
delq comm_q, g_q elt;
if (g_q_elt.nom = tempstr);
move #1, g_found;
endif;
addq g_q_elt, comm_q, #1;
sub g_q_size, #1, g_q_size;
endwhile;
unlock;
if (g_found = #1);
move g_q_elt.st_name, shared;
move g_q_elt.trc_name, sh_trc;
else;
move nil_string, shared;
move nil_string, sh_trc;
endif;
ENDPROC;

; PROCEDURE get_local (taskname : string)
; This procedure accepts a task name and sets the local and
; loc_trc variables to hold the names of the task's messaging and trace
; structures.
PROC get_local;
pop taskstk, tempstr;
move #0, g_found;
siseq loc_comm_q, g_q_size;
while (g_q_size > #0) & ("g_found");
delq loc_comm_q, g_q_elt;
if (g_q_elt.nom = tempstr);
move #1, g_found;
endif;
addq g_q_elt, loc_comm_q, #1;
sub g_q_size, #1, g_q_size;
endwhile;
if (g_found = #1);
move g_q_elt.st_name, local;
move g_q_elt.trc_name, loc_trc;
else;
move nil_string, local;
move nil_string, loc_trc;
endif;
ENDPROC;

; PRIMITIVE nilprim
; This dummy primitive does nothing except pull in internode
; messages.
PRIM nilprim;

426
disable $10;
swhile start_on;
move curr_tcbsts.rdy_time, crdy_str;
move curr_tcbsts.wt_time, Get_str;
move curr_tcbsts.run_time, Grun_str;
if (num_queued > 0);
move $0, done;
smove loc_comm, q, ss;
while (q > 0) & (done);
delq loc_comm, st_elt;
move st_elt.st_name, local;
call get_sh, st_elt.nom;
lock;
while ($shared.nummsgs > 0);
sub $1, num_queued;
sub $1, $shared.nummsgs;
delq $sharedmsgs, msg_elt;
if (Local.waiting = $1);
if (Local.wt_type = msg_elt.type);
addq msg_elt, Local.msg_q, $2;
while (msg_elt.tro_size > 0);
addq $sharedmsgs, tro_size;
addq tro_size, $msg_elt.tro_size;
addq $msg_elt.tro_size, $1, msg_elt.tro_size;
endwhile;
else;
addq msg_elt, Global.msg_q, $1;
while (msg_elt.tro_size > 0);
addq $sharedmsgs, tro_size;
addq tro_size, $msg_elt.tro_size;
addq $msg_elt.tro_size, $1, msg_elt.tro_size;
endwhile;
endif;
endif;
endwhile;
unlock;
addq st_elt, loc_comm, $1;
sub q, ss, $1, q, ss;
endif;
if (done = $1);
call sched_tr_task, Global.block;
endif;
endif;
enable $10;
ENDPRI;}
endif;
else:
  call response_update, msg_str, loc_trc, $1, $1;
  addq $msg_str, @local.msg_q, $1;
endif;
else:
call get_sh, dest_str;
lock:
  add $1, num_queued, num_queued;
  call response_update, msg_str, sh_trc, $1, $1;
  addq $msg_str, @shared.mags, $1;
  add $1, @shared.num_mags, @shared.num_mags;
unlock;
endif;

@time end_os;
@sub end_os, start_os, elapsed;
@add elapsed, os_ovhd, os_ovhd;

enable $10;

ENDPRIM;

; PRIM recv (msg_type : integer; @msg : string)
;
; This primitive performs a blocking selective receive operation.
; It accepts the type of message expected (0 means accept any type)
; and a pointer to where the message is to be placed. If a
; specific receive type is specified, the task is blocked until
; that type is available. Otherwise, the calling task is blocked
; until any message becomes available.

PRIM block_recv:

  @time curr_tcb.sts.finish;
  @sub curr_tcb.sts.finish, curr_tcb.sts.begin, curr_tcb.sts.begin;
  @add curr_tcb.sts.begin, curr_tcb.sts.run_time, curr_tcb.sts.run_time;
  @move curr_tcb.sts.run_time, @run_str;
  @time curr_tcb.sts.begin;
  move msg_type, @local.wt_type;
  move $1, @local.waiting;
  move msg_str, curr_tcb.alt_str;
  move #$wait/, curr_tcb.status;
  move taskstk, curr_tcb.stk;
  add $2, pc, curr_tcb.pcnt;
  move curr_tcb, @local.block;
  call run_trc_task;
  disable $10;
  move curr_tcb.alt_str, msg_str;
  call get_local, curr_task;
  move $0, @local.waiting;
  @iseq @local.msg_q, done;
  if (done > $0);
    delq @local.msg_q, @msg_str;
    call response_update, msg_str, loc_trc, $2;
  endif;
ENDPRIM;

PRIM recv:

  disable $10;
  @time start_os;
  pop taskstk, msg_str;
  pop taskstk, msg_type;
  call get_local, curr_task;
  call get_sh, curr_task;
  while (@shared.num_mags > $0);
    lock;
    @sub num_queued, $1, num_queued;
    @sub @shared.num_mags, $1, @shared.num_mags;
    @delq @shared.mags, msg_alt;
    @addq msg_alt, @local.msg_q, $1;
    @while (msg_alt.trc.size > $0);
      @delq @shared.trc, trace_alt;
      @addq trace_alt, @local.traces, $1;
      @sub msg_alt.trc.size, $1, msg_alt.trc.size;
    endwhile;
    unlock;
  endwhile;
  move $0, found;
  @iseq @local.msg_q, q.size;

428
if (q_size = $0);
call block_rcv;
else;
    move $1, found;
    if (msg_type <> $0);
        reorder msg queue because receive specific
        move null_queue, temp_q;
        move null_queue, temp2_q;
        move $0, found;
        sizeq @local.msg.q, m_num;
        while (m_num > $0);
            delq @local.msg.q, msg_elt;
            if (*found & (msg_elt.type = msg_type));
                move $1, found;
                addq msg_elt, temp_q, $2;
                while (msg_elt.trc_size > $0);
                    vdelq @local.traces, trace_elt;
                    addq trace_elt, templ_q, $2;
                    sub msg_elt.trc_size, %1, msg_elt.trc_size;
                endwhile;
                else;
                    addq msg_elt, temp_q, $1;
                    while (msg_elt.trc_size > $0);
                        vdelq @local.traces, trace_elt;
                        addq trace_elt, templ2_q, $1;
                        sub msg_elt.trc_size, %1, msg_elt.trc_size;
                    endwhile;
                endif;
        sizeq @local.msg.q, m_num;
    endwhile;
    move temp_q, @local.msg.q;
    move temp2_q, @local.traces;
    endif;
endif;
call block_rcv;
else;
delq @local.msg.q, msg_str;
move msg_str.prior, curr_tcb.priority;
call response_update, msg_str, loc_trc, $2;
    time end_os;
    sub end_os, start_os, elapsed;
    add elapsed, os_evhd, os_evhd;
endif;
endif;
enable $10;
ENDPRIM;

; PRIMITIVE sleep
; This primitive simply blocks the calling task by putting it in
; the sleep wait queue.
PRIM sleep:
    disable $10;
    *time start_os;
    pop taskstk, ret_pc;
    *time curr_tcb.sts.finish;
    sub curr_tcb.sts.finish, curr_tcb.sts.begin, curr_tcb.sts.begin;
    add curr_tcb.sts.begin, curr_tcb.sts.run_time, curr_tcb.sts.run_time;
    move $/wait/, curr_tcb.status;
    move tasksk, curr_tcb.stk;
    move ret_pc, curr_tcb.pc;
    addq curr_tcb, wait, curr_tcb.priority;
call run_rtr_task;
ENDPRIM;

; PRIMITIVE get_data (task_name : string; data_name : string; size : integer)
; This primitive simulates data access between tasks. If the
; specified source task is not resident on the processor, then a number
; of shared accesses are made. The access is made for every 100 bytes
; of data requested.
PRIM get_data:
    disable $10;

429
*time start_os;
pop taskstk, data_size;
pop taskstk, msg_str;
pop taskstk, dest_str;
call get_local, dest_str;
if (local = nil_string);
  while (data_size > 0);
    lock;
    move msg_str, q.size;
    unlock;
    sub data_size, $100, data_size;
  endwhile;
else;
  while (data_size > 0);
    move q.size, q.size;
    sub data_size, $100, data_size;
  endwhile;
endif;
*time end_os;
sub end_os, start_os, elapsed;
add elapsed, os_evhdl, os_evhdl;
enable $10;
ENDPRIM;

; PRIMITIVE end_resp
; This primitive simply calls response update to print the
; response traces.
PRIM end_resp;
  *disable $10;
  *pop taskstk, m_num;
  *call response_update, m_num, $6;
  *enable $10;
ENDPRIM;

; PRIMITIVE new_resp
; This primitive clears the calling task's current trace queue.
PRIM newresp;
  *disable $10;
  *seq curr_tcb.sts.resp_q, q.size;
  WHILE (q.size > 0);
    *delq curr_tcb.sts.resp_q, trace_elt;
    *sizeq curr_tcb.sts.resp_q, q.size;
  ENDWHILE;
  *enable $10;
ENDPRIM;

; PRIMITIVE evt_sync
; This primitive serves as the interrupt handler for the synchronization
; primitive. This interrupt is raised periodically every 5 seconds.
PRIM evt_sync;
  *disable $10;
  *add $1, evt_q, evt_q;
  *time start_os;
  *sizeq wait, wt_chk;
  move $0, found;
  while ((wt_chk > 0) & (found));
    *delq wait, temp_tcb;
    if (temp_tcb.name = #/synch/);
      move $1, found;
      move $70, temp_tcb.priority;
    else;
      *addq temp_tcb, wait, temp_tcb.priority;
      sub wt_chk, $1, wt_chk;
    endif;
  endwhile;
  if (found);
    *move curr_tcb, temp2_tcb;
    *move temp_tcb, curr_tcb;
    *call response_update, $2, $6;
    *move curr_tcb, temp_tcb;
    *move temp2_tcb, curr_tcb;
    call sched_str_task, temp_tcb;
  endif;
430
endif;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_ovhd, os_ovhd;

enable $10;
ENDPRIV;

; PRIMITIVE evt_loss_synch
;
; This primitive serves as the interrupt handler for the synchronization
; loss primitive. This interrupt is raised sporadically to signal a loss of
; synchronization.
PRIVM evt_loss_synch;
disable $10;
ackint $1;
*add $1, evt_i, evt_l;
*time start_os;
move $1, synch_loss;
sisq wait, wt_chk;
move #0, found;
while ((wt_chk > #0) & (~found));
delq wait, temp_tcb;
if (temp_tcb.name = #/synch/);
move wt, found;
move #0, temp_tcb.priority;
else;
addq temp_tcb, wait, temp_tcb.priority;
sub wt_chk, #1, wt_chk;
endif;
endwhile;
if (found);
*move curr_tcb, temp_tcb;
*move temp_tcb, curr_tcb;
*call response_update, $1, #0;
*move curr_tcb, temp_tcb;
*move temp_tcb, curr_tcb;
*call sched_rtr_task, temp_tcb;
endif;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_ovhd, os_ovhd;
enable $10;
ENDPRIV;

; PRIMITIVE evt_meg_rx
;
; This primitive serves as the interrupt handler for the incoming external
; message interrupt. This interrupt is raised sporadically to signal the
; arrival of a message. It makes the R50 task ready to run from the
; blocked receive state.
PRIVM evt_meg_rx;
disable $10;
ackint $2;
*add $1, evt_2, evt_2;
*time start_os;
move $1, msg_rdy;
rand $10, wt_chk;
if (wt_chk < #6);
move $1, msgtype;
else;
if (wt_chk < #6);
move $2, msgtype;
else;
move $3, msgtype;
endif;
endif;
move #0, done;
call get_local, #/msg/;
if (local.waiting = #1); move #0, local.waiting;
*move curr_tcb, temp_tcb;
*move local.block, curr_tcb;
*call response_update, $2, #0;
if (msgtype = #1); move #0, curr_tcb.priority;
else;
if (msgtype = #2);
move $80, curr_tcb.priority;
else;
move $50, curr_tcb.priority;
endif;
endif;
*move curr_tcb, @local.block;
*move temp2_tcb, curr_tcb;
move $1, done;
endif;
if (done = $1);
call sched_rtr_task, @local.block;
endif;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_ovhd, os_ovhd;

enable $10;

ENDPRIM;

; PRIMITIVE evt_signal
; This primitive serves as the interrupt handler for the signalling
; interrupt handler. This interrupt is raised sporadically to signal the
; arrival of a signalling event. It makes the SIGNAL task ready to run
; from the blocked receive state.

PRIM evt_signal;
disable $10;
ackint $3;
add $1, event_3, event_3;
*time start_os;
move $1, sig_rdy;
read $10, wt_chk;
if (wt_chk < $5);
move $6, sigtype;
else;
if (wt_chk < $3);
move $4, sigtype;
else;
move $5, sigtype;
endif;
endif;
move $0, done;
call get_local, #/signal/;
if (@local.waiting = $1);
move $0, @local.waiting;
*move curr_tcb, temp2_tcb;
*move @local.block, curr_tcb;
call response_update, $3, $6;
if (sigtype = $5);
move $80, curr_tcb.priority;
else;
if (sigtype = $6);
move $70, curr_tcb.priority;
endif;
endif;
*move curr_tcb, @local.block;
*move temp2_tcb, curr_tcb;
move $1, done;
endif;
if (done = $1);
call sched_rtr_task, @local.block;
endif;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_ovhd, os_ovhd;

enable $10;

ENDPRIM;

; PRIMITIVE evt_clk
; This primitive serves as the interrupt handler for the clock interrupt
; used as a timing base for the audits task. This is a periodic interrupt.

PRIM evt_clk;
disable $10;
ackint $4;
add $1, event_4, event_4;
*time start_os;
time seq start_seq, wt_chk;
move $0, found;
while ( (ws_chk > #0) & (*found) )
   { 
   defq wait, temp_tcb;
   if ( temp_tcb.name = #/audits/ )
      { 
      move #1, found;
      }
   else
      { 
      addq temp_tcb, wait, temp_tcb.priority;
      sub wait, #1, wait;
      endif;
      endwhile;
   
   if (*found)
      { 
      move #60, temp_tcb.priority;
      move temp_tcb, temp_tcb, curr_tcb;
      call response_update, #4, #6;
      move curr_tcb, temp_tcb;
      move temp_tcb, curr_tcb;
      call sched_evt_task, temp_tcb;
      endif;
      
      time end_os;
      sub end_os, start_os, elapsed;
      add elapsed, os_ovhd, os_prev;
      
      enable #10;
      
      ENDPN; }
move #/l_tii.traces/, st_elt.trc_name;
addq st_elt, loc_comm,q, #1;
move #/l_tii/, st_elt.st_name;
move #/l_tii.traces/, st_elt.trc_name;
addq st_elt, loc_comm,q, #1;
move #/l_tii3/, st_elt.st_name;
move #/l_tii3.traces/, st_elt.trc_name;
addq st_elt, loc_comm,q, #1;
move #/l_tii3/, st_elt.st_name;
move #/l_tii3.traces/, st_elt.trc_name;
addq st_elt, loc_comm,q, #1;
move #/l_tii4/, st_elt.st_name;
move #/l_tii4.traces/, st_elt.trc_name;
addq st_elt, loc_comm,q, #1;
move #/l_tii4/, st_elt.st_name;
move #/l_tii4.traces/, st_elt.trc_name;
addq st_elt, loc_comm,q, #1;
move #/l_tii5/, st_elt.st_name;
move #/l_tii5.traces/, st_elt.trc_name;
addq st_elt, loc_comm,q, #1;
move #/l_tii5/, st_elt.st_name;
move #/l_tii5.traces/, st_elt.trc_name;
addq st_elt, loc_comm,q, #1;

if (node_name = #/nodei/);

lock;
move #/sh_t1/, st_elt.st_name;
move #/sh_t1.trcs/, st_elt.trc_name;
addq st_elt, comm_q, #1;
move #/sh_t2/, st_elt.st_name;
move #/sh_t2.trcs/, st_elt.trc_name;
addq st_elt, comm_q, #1;
move #/sh_t3/, st_elt.st_name;
move #/sh_t3.trcs/, st_elt.trc_name;
addq st_elt, comm_q, #1;
move #/sh_t4/, st_elt.st_name;
move #/sh_t4.trcs/, st_elt.trc_name;
addq st_elt, comm_q, #1;
move #/sh_t5/, st_elt.st_name;
move #/sh_t5.trcs/, st_elt.trc_name;
addq st_elt, comm_q, #1;
move #/sh_t6/, st_elt.st_name;
move #/sh_t6.trcs/, st_elt.trc_name;
addq st_elt, comm_q, #1;
move #/sh_t6/, st_elt.st_name;
move #/sh_t6.trcs/, st_elt.trc_name;
addq st_elt, comm_q, #1;
move #/sh_t7/, st_elt.st_name;
move #/sh_t7.trcs/, st_elt.trc_name;
addq st_elt, comm_q, #1;
move #/sh_t8/, st_elt.st_name;
move #/sh_t8.trcs/, st_elt.trc_name;
addq st_elt, comm_q, #1;
move #/sh_t8/, st_elt.st_name;
move #/sh_t8.trcs/, st_elt.trc_name;
addq st_elt, comm_q, #1;
move #/sh_t9/, st_elt.st_name;
move #/sh_t9.trcs/, st_elt.trc_name;
addq st_elt, comm_q, #1;
move #/sh_t10/, st_elt.st_name;
move #/sh_t10.trcs/, st_elt.trc_name;
addq st_elt, comm_q, #1;
move #/sh_t11/, st_elt.st_name;
move #/sh_t11.trcs/, st_elt.trc_name;
addq st_elt, comm_q, #1;
move #/sh_t12/, st_elt.st_name;
move #/sh_t12.trcs/, st_elt.trc_name;
addq st_elt, comm_q, #1;
move #/sh_t13/, st_elt.st_name;
move #/sh_t13.trcs/, st_elt.trc_name;
addq st_elt, comm_q, #1;
move #/sh_t14/, st_elt.st_name;
move #/sh_t14.trcs/, st_elt.trc_name;
addq st_elt, comm_q, #1;
move #/sh_t15/, st_elt.st_name;
move #/sh_t15.trcs/, st_elt.trc_name;
addq st_elt, comm_q, #1;
move #/sh_t16/, st_elt.st_name;
move #/sh_t16.trcs/, st_elt.trc_name;
addq st_elt, comm_q, #1;
move #/sh_t17/, st_elt.st_name;
move #/sh_t17.trcs/, st_elt.trc_name;
addq st_elt, comm_q, #1;

unlock;

discard.

move tasklist, proclist;
sisec proclist, br_chk;
movq currtask, tempstr;
*mov $1, st_chk;

while br_chk <> #0;
dalq proclist, tcb_name;
cxsw tcb_name;

434
*time curr_tcb sts.begin;
*move ut chk, curr_tcb num;
*add $1, ut chk, ut chk;

if (tcb_name <> #/idle/)
lock
*typc nod_name;
delq loc_comm_q, st elt;
mov tcb_name, st elt nom;
*typc #/GLOBAL-- TASK: /;
*typc tcb_name;
*typc #/ QUEUE: /;
*typc st elt st name;
addq st elt, temp q, $1;
unlock;
lock;
delq comm q, st elt;
while (st elt nom <> nil string);
    addq st elt, comm q, $1;
delq comm q, st elt;
endwhile;
mov tcb_name, st elt nom;
*typc #/GLOBAL-- TASK: /;
*typc tcb_name;
*typc #/ QUEUE: /;
*typc st elt st name;
addq st elt, comm q, $1;
unlock;
endif;
mov taskstr, curr_tcb pnt;
mov tcb_name, curr_tcb name;
mov #/ready/, curr_tcb status;
mov currpric, curr_tcb priority;
mov taskstk, curr_tcb stck;
addq curr_tcb, rts, currpric;
siseq proclist, br chk;
endwhile;
mov temp q, loc_comm q;
mov nil_queue, temp q;
exit temp str;
mov #/exit tm/, exit str;
mov #/exit str/, exit str;
*mov #/END RESPONSE............./, end str;
bind e 1t_synch, $0;
bind e 1t_loss_synch, $1;
bind e 1t_msg_rx, $2;
bind e 1t_signal, $3;
bind e 1t_clk, $4;
bind e 1t_add, $5;

enable $10;
prnter #/Kernel Starting.../;
call run_xtr task;
printer #/ERROR - KERNEL TERMINATING/;
ENDTASK;
ENDKERNEL;
B.4.4 Response Priority Application Tasks

The following tasks were used with the above response priority kernel. The only differences between these tasks and those of section B.2.3 are that the activities of these tasks fill in the message tag field (msg.out.prior) before sending any messages. As such, only the activities affected by this change are included here. The task mainlines are exactly the same as those of section B.2.3.

```
<table>
<thead>
<tr>
<th>MSG Task</th>
</tr>
</thead>
</table>
| ACTIVITY send_entry;
| move $2, msg_out.type;
| move $0, msg_out.prior;
| move #/msg/, msg_out.src;
| ENDACT; |
| ACTIVITY send_start;
| move $1, msg_out.type;
| move $0, msg_out.prior;
| move #/msg/, msg_out.src;
| ENDACT; |
| ACTIVITY send_diag;
| move $2, msg_out.type;
| move $0, msg_out.prior;
| move #/msg/, msg_out.src;
| ENDACT; |
| SYNCH Task |
| ACTIVITY report_synch;
| move $0, synch lows;
| move $0, msg_out.type;
| move $0, msg_out.prior;
| move #/synch/, msg_out.src;
| ENDACT; |
| ACTIVITY change_synch;
| move $0, msg_out.type;
| move $0, msg_out.prior;
| move #/synch/, msg_out.src;
| ENDACT; |
| FAULT Task |
| ACTIVITY send_report;
| move $7, msg_out.type;
| move $0, msg_out.prior;
| move #/fault/, msg_out.src;
| ENDACT; |
| AUDITS Task |
| ACTIVITY tell_tmtc;
| move $23, msg_out.type;
| move $0, msg_out.prior;
| move #/audits/, msg_out.src;
| ENDACT; |
| ACTIVITY aud1;
| move $0, aud1 cnt;
| move $0, msg_out.prior;
| move $10, msg_out.type;
| move #/audits/, msg_out.src;
| ENDACT; |
| ACTIVITY aud2;
| move $0, aud2 cnt;
| move $10, msg_out.type;
| move $60, msg_out.prior;
| move #/audits/, msg_out.src;
| ENDACT; |
| ACTIVITY aud3;
| move $0, aud3 cnt;
| move $10, msg_out.type;
| move $60, msg_out.prior;
| move #/audits/, msg_out.src;
| ENDACT; |
| TMC Task |
| ACTIVITY report_fail;
| move $10, msg_out.type;
| move $60, msg_out.prior;
| move #/tmc/, msg_out.src;
| ENDACT; |
| DIAG Task |
| ACTIVITY chnl_req;
| move $12, msg_out.type;
| move $60, msg_out.prior;
| move #/diag/, msg_out.src;
| ENDACT; |
| ACTIVITY choose_diag;
| rand $4, dg_num;
| move $21, msg_out.type;
| move $60, msg_out.prior;
| move #/diag/, msg_out.src;
| if (dg_num == 0);
| move #/tmdiag/, dest;
| else;
| if (dg_num == 1);
| move #/chnldiag/, dest;
| else;
| if (dg_num == 2);
| move #/msdialg/, dest;
| else;
| move #/tdialg/, dest;
| endif;
| endif;
| ENDACT; |
| TIDiAG Task |
| ACTIVITY stop_mtc;
| move $21, msg_out.type;
| move $60, msg_out.prior;
| move #/tdialg/, msg_out.src;
| ENDACT; |
| ACTIVITY report_fail;
| move $10, msg_out.type;
| move $60, msg_out.prior;
| move #/tdialg/, msg_out.src;
| ENDACT; |
| MSGDIAG Task |
| ACTIVITY stop_integ;
| move $20, msg_out.type;
| move $60, msg_out.prior;
| ENDACT; |
move $#msgdiag, msg_out.src;
ENDACT;

ACTIVITY report_fail;
  move $10, msg_out.type;
  move $80, msg_out.prior;
  move $#/msgdiag, msg_out.src;
ENDACT;

; TONEDIAG Task

ACTIVITY select_tone;
  move $12, msg_out.type;
  move $80, msg_out.prior;
  move $#/tonediag, msg_out.src;
ENDACT;

ACTIVITY report_fail;
  move $10, msg_out.type;
  move $80, msg_out.prior;
  move $#/tonediag, msg_out.src;
ENDACT;

; CHELDIA Task

ACTIVITY connect_chnl;
  move $11, msg_out.type;
  move $80, msg_out.prior;
  move $#/cheldia, msg_out.src;
ENDACT;

ACTIVITY step_integ;
  move $20, msg_out.type;
  move $80, msg_out.prior;
  move $#/cheldia, msg_out.src;
ENDACT;

ACTIVITY report_fail;
  move $10, msg_out.type;
  move $80, msg_out.prior;
  move $#/cheldia, msg_out.src;
ENDACT;

; TONES Task

ACTIVITY select_tone;
  ticks $350;
  if (msg_in.src = $#/mag);
    move $1, msg_out.data;
  else;
    move $0, msg_out.data;
  endif;
  move $11, msg_out.type;
  move mag_in.prior, msg_out.prior;
  move $#/tones, msg_out.src;
ENDACT;

; CINIT Task

ACTIVITY chnl_req;
  move $12, msg_out.type;
  move $80, msg_out.prior;
  move $#/cinit, msg_out.src;
ENDACT;

ACTIVITY conn_req;
  move $11, msg_out.type;
  move $80, msg_out.prior;
  move $#/cinit, msg_out.src;
ENDACT;

ACTIVITY stat_req;
  move $16, msg_out.type;

; CHELS Task

ACTIVITY get_chnl;
  add $1, num_allocated, num_allocated;
  if (msg_in.src = $#/diag);
    add $1, mtoch, mtoch;
  else;
    add $1, cpch, cpch;
  endif;
  move $14, msg_out.type;
  move msg_in.prior, msg_out.prior;
  move $#/chels, msg_out.src;
ENDACT;

; STAT Task

ACTIVITY msg_call;
  update table
  ticks $200;
  move $1, stat_data;
  move $80, msg_out.prior;
  move $16, msg_out.type;
  move $#/stat, msg_out.src;
ENDACT;

ACTIVITY integ_fail;
  move $10, msg_out.type;
  move $80, msg_out.prior;
  move $1, stat_data;
  move $#/stat, msg_out.src;
ENDACT;

; END Task

ACTIVITY chnl_req;
  move $13, msg_out.type;
  move $70, msg_out.prior;
  move $#/end, msg_out.src;
ENDACT;

ACTIVITY conn_req;
  move $11, msg_out.type;
  move $70, msg_out.prior;
  move $#/end, msg_out.src;
ENDACT;

ACTIVITY stat_req;
  move $19, msg_out.type;
  move $70, msg_out.prior;
  move $#/end, msg_out.src;
ENDACT;

; CONNECT Task

All Activites the Same.

; SIGNAL Task

ACTIVITY send_stat;
  move $17, msg_out.type;
  move $80, msg_out.prior;
  move $#/signal, msg_out.src;
ENDACT;

ACTIVITY fin_stat;
  move $19, msg_out.type;
  move $70, msg_out.prior;

437
B.4.5 Shortest Task First Software

The kernel and application software used for this scheduling scheme is the same as that provided in section B.2.2 and B.2.3. A sample of the tasks used to determine task cycle times is presented below. This same type of task was used to determine cycle times for section B.4.6.

```
move $3, msg_out.type;
move msg_out.type, msg.out.prior;
move #/msg/, msg_out.src;
ENDACT;

ACTIVITY send_diag;
  move $3, msg_out.type;
  move $0, msg_out.prior;
  move $/signal/, msg_out.src;
ENDACT;

ACTIVITY send_diag;
  move $3, msg_out.type;
  move msg_out.type, msg_out.prior;
  move $/signal/, msg_out.src;
ENDACT;

ACTIVITY acc_in;
  stime tak_strt;
  if (msg_in.type <> $0);
    move $1, msg_pending;
  endif;
  ENDACT;

ACTIVITY clr_msg;
  move $0, msg_in.type;
  move $0, msg_out.type;
  move msg_out.type, msg_out.prior;
  move #/msg/, msg_out.src;
ENDACT;

ACTIVITY set_sync;
  sick $200;
  move $0, msg_in.type;
  move $0, msg_out.type;
  move $0, msg_pending;
ENDACT;

ACTIVITY time_upd;
  stime tak_end;
  sub tak_end, tak_start, tak_end;
  add tak_end, tot_time, tot_time;
ENDACT;

BEGIN
DO.ACT init;
  CALLP nilprim;
  SETSTATE $/clear/;
  LOOP:
    STATE_SELECT;
    STATE clear;
      IF #true;
        DO.ACT upd_cycle;
        CALLP newesp;
        SETSTATE $/start/;
      ENDIF;
      ENDSTATE;
    STATE start;
      IF (msg_pending = $0);
        CALLP nilprim;
        SETSTATE $/handl_msg/;
      ELSE;
        IF (msg_rdy = $1);
          DO.ACT inor_cnt;
          CALLP nilprim;
          SETSTATE $/xxtz/;
        ENDIF;
        IF #true;
          DO.ACT time_upd;
          CALLP recr $0, $msg_in/;
          SETSTATE $/xxtz/;
        ENDIF;
      ENDSTATE;
    STATE sigmsg;
      IF #true;
        DO.ACT ack_in;
        CALLP nilprim;
        SETSTATE $/#start/;
      ELSE;
        STATE xxtz;
          IF (byte_cnt < $32);
            DO.ACT inor_cnt;
            CALLP nilprim;
            SETSTATE $/xxtz/;
          ENDIF;
          IF #true;
            CALLP nilprim;
            SETSTATE $/msg_fin/;
          ENDIF;
          ENDSTATE;
        STATE msg_fin;
          IF (msg_type = $1);
```

438
DO_ACT send_start;
    CALLP send, $/chint/$, $/msg_out/$;
    SETSTATE $/clear/$;
ENDIF;
IF (msg_type = #2);
    DO_ACT send_tone;
    CALLP send, $/tones/$, $/msg_out/$;
    SETSTATE $/clear/$;
ENDIF;
IF (msg_type = #3);
    DO_ACT send_diag;
    CALLP send, $/diag/$, $/msg_out/$;
    SETSTATE $/clear/$;
ENDIF;
ENDSTATE;

STATE hnd: msg;
    IF (msg_in_type = #7);
        DO_ACT clr_msg;
        CALLP endresp, $1;
        SETSTATE $/clear/$;
    ENDIF;
    IF (msg_in_type = #6);
        DO_ACT set_synch;
        CALLP endresp, $2;
        SETSTATE $/clear/$;
    ENDIF;
ENDSTATE;
ENDSELECT;
ENDLOOP
ENDTASK;
B.4.6 Longest Task First Scheduling Software

The kernel and application software used for this scheduling scheme is the same as that provided in section B.2.2 and B.2.3.

B.4.7 Earliest Due Date Scheduling Kernel

```c
KERNEL #define
This kernel contains the following primitives:
nonblocking send
blocking selective receive
sleep
gut data from another task
killprim

The following external message types are supported:
1 - start call setup
2 - provide tone
3 - run diagnostic
4 - integrity ok
5 - integrity failure
6 - call takedown

The following internal message types are supported:
7 - START - request call setup
8 - TONES - request for tone
9 - DIAG - request single random diagnostic
10 - MSG - report fault
11 - MSG - perform and report synch operation
12 - FAULT - synch loss
13 - FAULT - general fault report
14 - CONNECT - perform connect/disconnect
15 - CBRXL - allocate channel
16 - CBRXL - free channel
17 - all tasks - channel granted
18 - STAT - call setup complete
19 - SIGNAL - start integrity checking
20 - STAT - integrity failure report
21 - EBD - caller hung up
22 - STAT - call finished
23 - STAT - stop integrity checking
24 - all diag - run diagnostic
25 - TINTC - stop carrier checking
26 - TINTC - monitor carriers

The following response types are supported:
81 - MSG - fault report sent to switching center
82 - MSG - synch source changed
83 - CONNECT - switching center tone request connected
84 - STAT - call finished
85 - STAT - call started
86 - alldiag - diagnostic completed without failure
87 - TINTC - carrier scan completed without failure

KERNEL baseb;
TASK kern_task, 50;
TYPE t_tcb_stats = rdy_time : integer *;
run_time : integer *;
est_time : integer *;
begin : integer *;
finish : integer *;
resp_q : queue ; of t_trace_elt

TYPE t_shared_stats = data_trace : queue ; of t_trace_elt

TYPE t_tcb = name : string * ; task name
num : integer * ;
status : string * ; task state
priority : integer * ; priority
pcnt : integer * ; program counter
elt_ent: string * ; rcv msg elt
stk : stack * ; local task stack

440

st : t_tcb_state ; task control block

TYPE t_sh_data = task_num : string *;  
task name
   num_mgq : integer *;  
   number of queued messages
   mgq : queue *;  
   message queue (t_msg_elt)
   traces : queue *;  
   trace queue (t_trace_elt)

TYPE t_msg_struct = task_num : string *;  
task name
   waiting : integer *;  
   flag set if blocked for msg
   mgq : queue *;  
   message queue (t_msg_elt)
   traces : queue *;  
   trace queue (t_trace_elt)
   mg_type : integer *;  
   type of msg blocked on
   block : t_tcb;  
   tcb wait area

TYPE t_id_struct = ncm : string *;  
task name
   tcb_name : string *;  
st_name : string;

SHARED comm_q, queue;  
of t_id_struct (one elt per task)
SHARED sh_h1, t_sh_data;
SHARED sh_h2, t_sh_data;
SHARED sh_h3, t_sh_data;
SHARED sh_h4, t_sh_data;
SHARED sh_h5, t_sh_data;
SHARED sh_h6, t_sh_data;
SHARED sh_h7, t_sh_data;
SHARED sh_h8, t_sh_data;
SHARED sh_h9, t_sh_data;
SHARED sh_h10, t_sh_data;
SHARED sh_h11, t_sh_data;
SHARED sh_h12, t_sh_data;
SHARED sh_h13, t_sh_data;
SHARED sh_h14, t_sh_data;
SHARED sh_h15, t_sh_data;
SHARED sh_h16, t_sh_data;
SHARED sh_h17, t_sh_data;
SHARED ti_data, integer;
SHARED smsg_data, integer;
SHARED num_queueed, integer;

; Message queues
VAR loc_comm, queue;  
of t_id_struct (one elt per local task)
VAR st_elt, t_id_struct;
VAR 1_t1, t_msg_struct;
VAR 1_t2, t_msg_struct;
VAR 1_t3, t_msg_struct;
VAR 1_t4, t_msg_struct;
VAR 1_t5, t_msg_struct;
VAR 1_t6, t_msg_struct;
VAR 1_t7, t_msg_struct;
VAR 1_t8, t_msg STRUCT;
VAR 1_t9, t_msg STRUCT;
VAR 1_t10, t_msg STRUCT;
VAR 1_t11, t_msg STRUCT;
VAR 1_t12, t_msg STRUCT;
VAR 1_t13, t_msg STRUCT;
VAR 1_t14, t_msg STRUCT;
VAR 1_t15, t_msg STRUCT;
VAR 1_t16, t_msg STRUCT;
VAR local, string;
VAR shared, string;
VAR loc_tcb, string;
VAR sh_tcb, string;

STAT sends, integer;
STAT os_cvhdl, integer;
STAT sc_cvhdl, integer;
STAT switches, integer;
STAT preempts, integer;
STAT evr_0, integer;
STAT evr_1, integer;
STAT evr_2, integer;
STAT evr_3, integer;
STAT evr_4, integer;
STAT evr_5, integer;
VAR start_os, integer;
VAR end_os, integer;
VAR elapsed, integer;
VAR os_start, integer;
VAR os_end, integer;

441
VAR rdy_str, string;
VAR wt_str, string;
VAR run_str, string;
VAR end_str, string;
VAR swait, queue;
VAR wt_chk, integer;
VAR msg_str, string;
VAR dest_str, string;
VAR tsp_tcb, t_tcb;
VAR temp_tcb, t_tcb;
VAR ret_pc, integer;
VAR tmp_eblk, stack;
VAR num_parms, integer;
VAR found, integer;
VAR msg_type, integer;
VAR data_size, integer;
VAR nil_queue, queue;
VAR nil_string, string;
VAR msg_elts, t_msg_elts;
VAR m_num, integer;
VAR temp_q, queue;
VAR typ, integer;
VAR q, integer;
VAR done, integer;
VAR ch_syn_due, integer;
VAR l_syn_due, integer;
VAR msg_due, integer;
VAR sig_due, integer;
VAR end_due, integer;
VAR t_prio, integer;

; Global Kernel Variables
VAR curr_tcb, t_tcb ; current task control block
VAR proclist, queue ; copy of task name list
VAR bcb, integer ; branch condition check
VAR tempstr, string ; temporary string value
VAR tcb_name, string ; current task name
VAR rtr, queue ; ready to run queue

; Variables for get_local and get_sh procedures
VAR g_q_size, integer;
VAR g_found, integer;
VAR g_st_elts, t_id_stack;

; Variables for response_update procedure
VAR action, integer ; action requested in response_update
VAR prio, integer ; send priority
VAR trace_elts, t_trace_elts ; new response trace element
VAR msg_elts_name, string ; name of msg elt for response trace
VAR dest_elts_name, string ; name of msg queue
VAR delim_elts, t_trace_elts ; msg or data trace delimiter
VAR q_size, integer ; size of trace queue
VAR data_name, string ; name of shared data variable
VAR temp_q, queue ; data response queue copy

; VARIABLE declarations for procedure RUN_RTR_TASK
VAR error, boolean; ; Set if no ready processes

; VARIABLE declarations for procedure SCHED_RTR_TASK
VAR rtr_pcb, t_tcb; ; PFC to be scheduled

; PROCEDURE response_update (Gname : string;
; action: integer)
; This procedure updates the response trace queues
PROC response_update;
*pop taskstk, action;
*move nod_name, trace_elts.node_name;
*move curr_tcb_name, trace_elts.task_name;
*move trace_elts.time;
*move action, trace_elts.act;

442
*if (action = #1);  SEND MSG
  *pop taskstk, dest_e1t_name;
  *pop taskstk, msg_e1t_name;
  *addq trace_e1t, curr_tcb.sts.resp_q, @1;
  *move curr_tcb.sts.resp_q, temp_q;
  *sizeq temp_q, q_size;
  *move q_size, msg_e1t_name, trace_e1t;
  *while (q_size > 80);
    *delq temp_q, trace_e1t;
    *addq trace_e1t, dest_e1t_name, temp_q;
    *sizeq temp_q, q_size;
  *endwhile;
*endif;

*if (action = #2);  RCV MSG
  *pop taskstk, dest_e1t_name;
  *pop taskstk, msg_e1t_name;
  *move trace_e1t, delim_e1t;
  *addq trace_e1t, curr_tcb.sts.resp_q, @1;
  *move @1, delim_e1t, act;
  *addq delim_e1t, curr_tcb.sts.resp_q, @1;
  *while (q_size > 80);
    *delq trace_e1t, msg_e1t_name, trace_e1t;
    *addq trace_e1t, curr_tcb.sts.resp_q, @1;
    *sub q_size, @1, q_size;
  *endwhile;
  *move @10, delim_e1t, act;
  *addq delim_e1t, curr_tcb.sts.resp_q, @1;
*endif;

*if (action = #3);  DATA WRITE
  *pop taskstk, data_name;
  *addq trace_e1t, curr_tcb.sts.resp_q, @1;
  *move curr_tcb.sts.resp_q, data_name, status.trace_q;
*endif;

*if (action = #4);  DATA READ
  *pop taskstk, data_name;
  *move trace_e1t, delim_e1t;
  *addq trace_e1t, curr_tcb.sts.resp_q, @1;
  *move @7, delim_e1t, act;
  *addq delim_e1t, curr_tcb.sts.resp_q, @1;
  *move @data_name, status.data_trace, temp_q;
  *sizeq temp_q, q_size;
  *while (q_size > 80);
    *delq temp_q, trace_e1t;
    *addq trace_e1t, curr_tcb.sts.resp_q, @1;
    *sizeq temp_q, q_size;
  *endwhile;
  *move @8, delim_e1t, act;
  *addq delim_e1t, curr_tcb.sts.resp_q, @1;
*endif;

*if (action = #5);  RESPONSE OUT
  *pop taskstk, trace_e1t.misc1;
  *addq trace_e1t, curr_tcb.sts.resp_q, @1;
  *sizeq curr_tcb.sts.resp_q, q_size;
  *while (q_size > 80);
    *delq curr_tcb.sts.resp_q, trace_e1t;
  *etype #/task;
  *etype trace_e1t, taskname;
  *etype #/action;
  *etype trace_e1t, act;
  *etype #/time;
  *etype trace_e1t, time;
  *etype #/misc1;
  *etype trace_e1t, misc1;
  *etype #/misc2;
  *etype trace_e1t, misc2;
  *sizeq curr_tcb.sts.resp_q, q_size;
  *endwhile;
  *etype trace_e1t, end_str;
*endif;

*if (action = #6);  EVERY TRIGGER
  *pop taskstk, trace_e1t.misc1;
  *etime trace_e1t, misc1, trace_e1t, misc2;

443
PROCEDURE run_rtr_task

This procedure takes the first ready to run task off the rtr queue (this is the one with the highest priority) and copies its saved state into the current processor state. If there are no ready to run tasks, then an error message is printed and the kernel terminates.

PROC run_rtx_task;
  *time sc_start;
  nseq rtr, br_chk;
  IF (br_chk = SO);
    printer */ERROR: NO ATR PROCESSES/;
    halt;
  ELSE;
    dq rtr, curr_tcb;
    move curr_tcb.name, tcb_name;
    ctest tcb_name;
    *add $1, switches, switches;
    *time curr_tcb.sts.finish;
    *sub curr_tcb.sts.finish, curr_tcb.sts.begin, curr_tcb.sts.begin;
    *add curr_tcb.sts.begin, curr_tcb.sts.xdy_time, curr_tcb.sts.xdy_time;
    *move */rdy_tm/, rdy_str;
    *move curr_tcb.sts.xdy_time, rdy_str;
    *move curr_tcb.sts.xdy_time, curr_tcb.sts.begin;
    *time curr_tcb.sts.begin;
    move */running/, curr_tcb.status;
    move curr_tcb.stk, taskstk;
    *time end_os;
    *sub end_os, start_os, elapsed;
    *add elapsed, os_evh, os_evh;
    *time sc_end;
    *sub sc_end, sc_start, sc_start;
    *add sc_start, sc_evh, sc_evh;
    enable #10;
    move curr_tcb.pcnt, pc;
  ENDIF;
ENDPROC;

PROCEDURE sched_rtr_task (rtr_tcb : t_tcb)

This procedure accepts a process control block and adds it to the ready to run queue in a prioritised fashion.

PROC sched_rtx_task;
  *time sc_start;
  pop taskstk, rtr_tcb;
  move */ready/, rtr_tcb.status;
  *time rtr_tcb.sts.finish;
  *sub rtr_tcb.sts.finish, rtr_tcb.sts.begin, rtr_tcb.sts.begin;
  *add rtr_tcb.sts.begin, rtr_tcb.sts.xdy_time, rtr_tcb.sts.xdy_time;
  *move taskstk, tmp_stk;
  *time rtr_tcb.sts.begin;
  *sub */70000, rtr_tcb.priority, t_prio;
  addq rtr_tcb, rtr_tcb.prio;
  if (rtr_tcb.priority < curr_tcb.priority); PRIORITY IS ACTUALLY DUE LATE
  pop taskstk, rst_pc;
  pop taskstk, rst_tcb;
  move rst_pc, curr_tcb.pcnt;
  move taskstk, curr_tcb.stk;
  *time curr_tcb.sts.finish;
  *sub curr_tcb.sts.finish, curr_tcb.sts.begin, curr_tcb.sts.begin;
  *add curr_tcb.sts.begin, curr_tcb.sts.run_time, curr_tcb.sts.run_time;
  *move curr_tcb.sts.run_time, curr_tcb.sts.begin;
  *time curr_tcb.sts.begin;
  move */ready/, curr_tcb.status;
  addq curr_tcb, rtr_tcb.prio;
  *time #1, preempt, preempt;
  *time sc_end;
  *sub sc_end, sc_start, sc_start;

444
@add sc_start, sc_ovhd, sc_ovhd;
call run_rts_task;
halt;
endif;
etime sc_end;
@end sc_end, sc_start, sc_start;
@end sc_start, sc_ovhd, sc_ovhd;

; PROCEDURE get_sh (taskname : string)
; This procedure accepts a task name and sets the shared and
; sh_trc variables to hold the names of the task's messaging and trace
; structures.
PROC get_sh;
      pop taskstk, tempstr;
      move $0, g_found;
      lock;
      sizeq comm_q, g_q_size;
      while (g_q_size > $0) & (¬g_found);
         delq comm_q, g_st_alt;
         if (g_st_alt.nom = tempstr);
            move $1, g_found;
         endif;
         addq g_st_alt, comm_q, $1;
         sub g_q_size, $1, g_q_size;
      endwhile;
      unlock;
      if (g_found = $1);
         move g_st_alt_st_name, shared;
         move g_st_alt_trc_name, sh_trc;
      else;
         move nil_string, shared;
         move nil_string, sh_trc;
      endif;
ENDPROC;

; PROCEDURE get_local (taskname : string)
; This procedure accepts a task name and sets the local and
; loc_trc variables to hold the names of the task's messaging and trace
; structures.
PROC get_local;
      pop taskstk, tempstr;
      move $0, g_found;
      sizeq loc_comm_q, g_q_size;
      while (g_q_size > $0) & (¬g_found);
         delq loc_comm_q, g_st_alt;
         if (g_st_alt.nom = tempstr);
            move $1, g_found;
         endif;
         addq g_st_alt, loc_comm_q, $1;
         sub g_q_size, $1, g_q_size;
      endwhile;
      if (g_found = $1);
         move g_st_alt_st_name, local;
         move g_st_alt_trc_name, loc_trc;
      else;
         move nil_string, local;
         move nil_string, loc_trc;
      endif;
ENDPROC;

; PRIMITIVE nilprim
; This dummy primitive does nothing except pull in internode
; messages.
PRIM nilprim;
   disable $10;
   *time start_os;
   *move cur_tcb.uts.rdy_time, &rdy_str;
   *move cur_tcb.uts.st_time, &cur_str;
   *move cur_tcb.uts.run_time, &run_str;
   if (num_queued > $0);
      move $0, done;
      sizeq loc_comm_q, q_es;
   endif;

445
while (q.sz > 0) & (*done);
    delq loc_comm.q, sv_elt;
    move sv_elt.el_name, local;
    call get_sh, sv_elt.nom;
    lock;
    while (shared.num_msgs > 0);
        sub num_queued, $1, num_queued;
        sub shared.num_msgs, $1, shared.num_msgs;
        delq sharedmsgs, msg_elt;
    if (local.waiting = $1);
        if (local.wt_type = msg_elt.type);
            addq msg_elt, loc_msg.q, #2;
            while (msg_elt.trc_size > 0);
                delq shared.trc.q, trace_elt;
                addq trace_elt, loc_trace.q, #2;
                sub msg_elt.trc_size, $1, msg_elt.trc_size;
            endwhile;
            else;
                addq msg_elt, loc_msg.q, #1;
                while (msg_elt.trc_size > 0);
                delq shared.trc.q, trace_elt;
                addq trace_elt, loc_trace.q, #1;
                sub msg_elt.trc_size, $1, msg_elt.trc_size;
            endwhile;
        endif;
        endif;
    unlock;
    addq sv_elt, loc_comm.q, #1;
    sub q.sz, $1, q.sz;
    endwhile;
    if (done = $1);
        call sched_xtr_task, local.block;
    endif;
    endif;
    enable #10;
ENDPRIM;

; PRIMITIVE send(Task_name : string; M msg : string)
; This primitive accepts the name of a destination task and a
; message pointer. It then performs a nonblocking send operation.
PRIM send;
    disable #10;
    add $1, sends, sends;
    *time start_os;
    pop tasksk, msg_str;
    pop tasksk, dest_str;
    call get_local, dest_str;
    if (local <> nil_string);
        if (local.waiting = $1);
            if (local.wt_type = msg_str.type);
                call response_update, msg_str, loc_trc, #2, $1;
                addq msg_str, loc_msg.q, #2;
            else;
                call response_update, msg_str, loc_trc, $1, #1;
                addq msg_str, loc_msg.q, #1;
            endif;
        endif;
        if ((local.wt_type = $0) | (local.wt_type = msg_str.type));
            move $0, local.waiting;
            move msg_str.prior, local.block.priority;
            call sched_xtr_task, local.block;
        endif;
        else;
            call response_update, msg_str, loc_trc, $1, #1;
            addq msg_str, loc_msg.q, #1;
        endif;
        else;
            call get_sh, dest_str;
    lock;
    add $1, num_queued, num_queued;

446
call response_update, msg_str, sh_trc, #1, #1;
add @msg_str, @shared.mags, #1;
add #1, @shared.num_mags, @shared.num_mags;
unlock;
endif;

*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, oscvhd, oscvhd;

enable $10;

ENDPRIM;

PRIM recv (msg_type : integer; msg : string)

This primitive performs a blocking selective receive operation.
It accepts the type of message expected (#0 means accept any type)
and a pointer to where the message is to be placed. If a
specific receive type is specified, the task is blocked until
that type is available. Otherwise, the calling task is blocked
until any message becomes available.

PROC block_recv;
*time curr_tcb sts.finish;
*sub curr_tcb sts.finish, curr_tcb sts.begin, curr_tcb sts.begin;
*add curr_tcb sts.begin, curr_tcb sts.run_time, curr_tcb sts.run_time;
move curr_tcb sts.run_time,ometown_str;
*time curr_tcb sts.begin;
move msg_type, @local.w_type;
move #1, @local.waiting;
move msg_str, curr_tcb elt_str;
move $/wait/, curr_tcb status;
move taskstk, curr_tcb stk;
add #2, pc, curr_tcb ptr;
move curr_tcb @local.block;
call run_str task;
disable $10;
move curr_tcb elt_str, msg_str;
call get_local, curr_task;
move #0, @local.waiting;
sizeq @local.msg_q, done;
if (done > #0);
delq @local.msg_q, msg_str;
call response_update, msg_str, loc_trc, #2;
endif;

ENDPROC;

PRIM recv;
disable $10;
*time start_os;
pop taskstk, msg_str;
pop taskstk, msg_type;
call get_local, curr_task;
call get_sh, curr_task;
while (@shared.num_mags > #0);
lock;
sub num_queued, #1, num_queued;
sub @shared.num_mags, #1, @shared.num_mags;
delq @shared.mags, msg elt;
addq msg elt, @local.msg_q, #1;
while (msg elt trc size > #0);
*delq @shared.trc, trace elt;
addq trace elt, @local.traces, #1;
sub msg elt trc size, #1, msg elt trc size;
endwhile;
unlock;
endwhile;
move #0, found;
sizeq @local.msg_q, q size;
if (q size = #0);
call block_recv;
else;
move #1, found;
if (msg type <> #0);
reorder msg queue because specific
move nil_queue, temp_q;

447
*move nil_queue, temp2_q;
move #0, found;
siszq local_msg.q, m_num;
while (m_num > #0);
delq [local.msg.q, msg.elts];
if (*found) & (msg.elts.type = msg.type):
move #1, found;
addq msg.elts, temp_q, #2;
while (msg.elts.trc.size > #0);
+delq [local.traces, trace_elts];
+addq trace_elts, temp2_q, #2;
-sub msg.elts.trc.size, #1, msg.elts.trc.size;
*endwhile;
else;
addq msg.elts, temp_q, #1;
*while (msg.elts.trc.size > #0);
+delq [local.traces, trace.elts];
+addq trace_elts, temp2.q, #2;
-sub msg.elts.trc.size, #1, msg.elts.trc.size;
*endwhile;
endif;
siszq local.msg.q, m_num;
*endwhile;
move temp.q, local.msg.q;
*move temp2.q, local.traces;
*endif;
if (*found);
call block_rcv;
else;
+delq [local.msg.q, msg_str];
mov msg_str.prior, curr_tcb.priority;
call response_update, msg_str; loc_tcb, #2;
+time end.os;
-sub end.os, start.os, elapsed;
+add elapsed, os_ovhd, os_ovhd;
endif;
*enable #10;
ENDPRIM;

; PRIMITIVE sleep
; This primitive simply blocks the calling task by putting it in
; the sleep wait queue.
PRIM sleep;
disable #10;
+time start.cs;
pop taskstk, ret_pc;
+time curr_tcb.sts.end;
+sub curr_tcb.sts.end, curr_tcb.sts.begin, curr_tcb.sts.begin;
+add curr_tcb.sts.begin, curr_tcb.sts.run_time, curr_tcb.sts.run_time;
+move curr_tcb.sts.run_time, @run_str;
+time curr_tcb.sts.begin;
move #/wait, curr_tcb.status;
move taskstk, curr_tcb stk;
move ret_pc, curr_tcb.pcnt;
+add curr_tcb, wait, curr_tcb.priority;
call run_tcb_task;
ENDPRIM;

; PRIMITIVE get_data (task_name : string; data_name : string; size : integer)
; This primitive simulates data access between tasks. If the
; specified source task is not resident on the processor, then a number
; of shared accesses are made. One access is made for every 100 bytes
; of data requested.
PRIM get_data;
disable #10;
+time start.cs;
pop taskstk, data_size;
pop taskstk, msg_str;
pop taskstk, dest_str;
call get_local, dest_str;
if (local = nil_string);
while (data.size > #0);
lock;

448
move msg_str, q_size;
unlock;
sub data_size, #100, data_size;
endwhile;
else:
while (data_size > #0);
move q_size, q_size;
sub data_size, #100, data_size;
endwhile;
endif;

*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os.ovhd, os_ovhd;
enable $10;

ENDPRIM;

; PRIMITIVE end_resp
; This primitive simply calls response.update to print the
; response traces.
PRIM end.resp;
*disable $10;
*pop taskstk, m_num;
*call response_update, m_num, #5;
*enable $10;

ENDPRIM;

; PRIMITIVE new Resp
; This primitive clears the calling task's current trace queue.
PRIM new.resp;
*disable $10;
*sizeq curr_tcb.sts.resp_q, q_size;
*WHILE (q_size > #0);
*delq curr_tcb.sts.resp_q, trace_elt;
*sizeq curr_tcb.sts.resp_q, q_size;
*ENDWEND;
*enable $10;

ENDPRIM;

; PRIMITIVE evt_synch
; This primitive serves as the interrupt handler for the synchronization
; primitive. This interrupt is raised periodically every 5 seconds.
PRIM evt_synch;
*disable $10;
*clear int #0;
*add $1, evt_O, event;
*time start_os;
*sizeq wait, wt_chk;
move #0, found;
while ((wt_chk > #0) & ("found"));
*delq wait, temp_tcb;
if ((temp_tcb.name = "$/synch"));
move #1, found;
time ch_syn_due;
*add $500, ch_syn_due, ch_syn_due;
move ch_syn_due, temp_tcb.priority;
else;
*add temp_tcb, wait, temp_tcb.priority;
*sub wt_chk, #1, wt_chk;
endif;
endwhile;
if ("found")
*move curr_tcb, temp_tcb;
*move temp_tcb, curr_tcb;
*call response_update, $2, #5;
*move curr_tcb, temp_tcb;
*move temp_tcb, curr_tcb;
call sched_trx_task, temp_tcb;
endif;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_ovhd, os_ovhd;
enable $10;

ENDPRIM;
PRIMITIVE evt_loss_synch

This primitive serves as the interrupt handler for the synchronization
loss primitive. This interrupt is raised sporadically to signal a loss of
synchronization.

PRIM evt_loss_synch;
    disable $10;
    mackit $1;
    *add $1, evt_1, evt_1;
    *time start_os;
    move $1, synch_loss;
    seteq wait, wt_chk;
    move $0, found;
    while ((wt_chk > $0) & !"found");
    delay wait, temp_tcb;
    if (temp_tcb.name = "$/synch");
        move $1, found;
        time l_syn_due;
        add $700, l_syn_due, l_syn_due;
        move l_syn_due, temp_tcb.prior;
    else;
        add temp_tcb, wait, temp_tcb.prior;
        sub wt_chk, $1, wt_chk;
    endif;
    endwhile;

    if (!found);
        move curr_tcb, temp_tcb;
        move temp_tcb, curr_tcb;
        *call response_update, $1, $6;
        move curr_tcb, temp_tcb;
        move temp_tcb, curr_tcb;
        *call sched_rte_task, temp_tcb;
    endif;
    *time end_os;
    *add end_os, start_os, elapsed;
    *add elapsed, os_ovhd, os_ovhd;
    enable $10;
ENDPRIM;

PRIMITIVE evt_msg_rx

This primitive serves as the interrupt handler for the incoming external
message interrupt. This interrupt is raised sporadically to signal the
arrival of a message. It makes the MSG task ready to run from the
blocked receive state.

PRIM evt_msg_rx;
    disable $10;
    mackit $2;
    *add $1, evt_2, evt_2;
    *time start_os;
    move $1, msg_rdy;
    rand $10, wt_chk;
    if (wt_chk < $6);
        move $1, msgtype;
        else;
            if (wt_chk < $3);
                move $2, msgtype;
            else;
                move $3, msgtype;
            endif;
        endif;
    move $0, done;
    call get_local, $/msg/;
    if (@local.waiting = $1);
        move $0, @local.waiting;
        move curr_tcb, temp_tcb;
        move @local.block, curr_tcb;
        *call response_update, $2, $6;
        time msg_due;
        if (msgtype = $1);
            add $3000, msg_due, msg_due;
            move msg_due, curr_tcb.prior;
        else;
            if (msgtype = $2);
                add $3000, msg_due, msg_due;
                move msg_due, curr_tcb.prior;
            else;
    endif;
add $12000, msg_due, msg_due;
move msg_due, curr_tcb.priority;
endif;
endif;
*move curr_tcb, @local.block;
*move temp_tcb, curr_tcb;
move $1, done;
endif;
if (done = $1);
call sched_rtr_task, @local.block;
endif;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_ovhd, os_ovhd;

enable $10;
ENDPRIM;

; PRIMITIVE evt_signal
;
; This primitive serves as the interrupt handler for the signalling
; interrupt handler. This interrupt is raised sporadically to signal the
; arrival of a signalling event. It makes the SIGNAL task ready to run
; from the blocked receive state.

PRIM evt_signal;
disable $10;
ackint $8;
*add $1, evt_3, evt_3;
*time start_os;
move $1, sig_rdy;
rand $10, ut_chk;
if (ut_chk < #6);
move $8, sigtype;
else;
if (ut_chk < #6);
move $4, sigtype;
else;
move $5, sigtype;
endif;
endif;
move $0, done;
call get_local, #/signal/;
if (@local.waiting = $1);
move $0, @local.waiting;
*move curr_tcb, temp_tcb;
*move @local.block, curr_tcb;
*call response_update, $3, $6;

time sig_due;
if (sigtype = $6);
*add $1000, sig_due, sig_due;
move sig_due, curr_tcb.priority;
else;
if (sigtype = $0);
*add $700, sig_due, sig_due;
move sig_due, curr_tcb.priority;
else;
*add $2000, sig_due, sig_due;
move sig_due, curr_tcb.priority;
endif;
*move curr_tcb, @local.block;
*move temp_tcb, curr_tcb;
move $1, done;
endif;
if (done = $1);
call sched_rtr_task, @local.block;
endif;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_ovhd, os_ovhd;

enable $10;
ENDPRIM;

; PRIMITIVE evt_clk
;
; This primitive serves as the interrupt handler for the clock interrupt
; used as a timing base for the audits task. This is a periodic interrupt.

451
PRIM evt_clk;

disable #10;
*ackint #4;
*add #1, evt_4, evt_4;
*time start_os;
siseq wait, wt_clk;
move #0, found;
while ((wt_clk > #0) & (*found));
delq wait, temp_tcb;
if (temp_tcb.name = #/audits/);
move #1, found;
else
add temp_tcb, wait, temp_tcb.priority;
sub wt_clk, #1, wt_clk;
endif;
endwhile;
if (*found);
time aud_due;
add #100, aud_due, aud_due;
move aud_due, temp_tcb.priority;
move curr_tcb, temp_tcb;
move temp_tcb, curr_tcb;
call response_update, #4, #6;
move curr_tcb, temp_tcb;
move temp_tcb, curr_tcb;
call sched_rtr_task, temp_tcb;
endif;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_vbd, os_vbd;
*senable #10;
ENDPRIM;

BEGIN
move #/node2/, nod_name;
disable #10;
move #/l_t1/, st_elt-st_name;
move #/l_t1.traces/, st_elt-trc_name;
addq st_elt, loc_comm_q, #1;
move #/l_t2/, st_elt-st_name;
move #/l_t2.traces/, st_elt-trc_name;
addq st_elt, loc_comm_q, #1;
move #/l_t3/, st_elt-st_name;
move #/l_t3.traces/, st_elt-trc_name;
addq st_elt, loc_comm_q, #1;
move #/l_t4/, st_elt-st_name;
move #/l_t4.traces/, st_elt-trc_name;
addq st_elt, loc_comm_q, #1;
move #/l_t5/, st_elt-st_name;
move #/l_t5.traces/, st_elt-trc_name;
addq st_elt, loc_comm_q, #1;
move #/l_t6/, st_elt-st_name;
move #/l_t6.traces/, st_elt-trc_name;
addq st_elt, loc_comm_q, #1;
move #/l_t7/, st_elt-st_name;
move #/l_t7.traces/, st_elt-trc_name;
addq st_elt, loc_comm_q, #1;
move #/l_t8/, st_elt-st_name;
move #/l_t8.traces/, st_elt-trc_name;
addq st_elt, loc_comm_q, #1;
move #/l_t9/, st_elt-st_name;
move #/l_t9.traces/, st_elt-trc_name;
addq st_elt, loc_comm_q, #1;
move #/l_t10/, st_elt-st_name;
move #/l_t10.traces/, st_elt-trc_name;
addq st_elt, loc_comm_q, #1;
move #/l_t11/, st_elt-st_name;
move #/l_t11.traces/, st_elt-trc_name;
addq st_elt, loc_comm_q, #1;
move #/l_t12/, st_elt-st_name;
move #/l_t12.traces/, st_elt-trc_name;
addq st_elt, loc_comm_q, #1;
move #/l_t13/, st_elt-st_name;
move #/l_t13.traces/, st_elt-trc_name;
addq st_elt, loc_comm_q, #1;
move #/l_t14/, st_elt-st_name;
move #/l_t14.traces/, st_elt-trc_name;
addq st_elt, loc_comm_q, #1;

452
move #/l.ti6, st elt.st_name;
addq st elt, loc.comm.q, $1;
move #/l.ti6.trace/, st elt.trace_name;
addq st elt, loc.comm.q, $1;
move #/l.ti6/ , st elt.st_name;
move #/l.ti6.trace/, st elt.trace_name;
addq st elt, loc.comm.q, $1;
if (nod.name = #/node1/)
    lock:
        move #/sh.ti1/, st elt.st_name;
        move #/sh.ti1.trace/, st elt.trace_name;
        addq st elt, loc.comm.q, $1;
        move #/sh.ti2/, st elt.st_name;
        move #/sh.ti2.trace/, st elt.trace_name;
        addq st elt, loc.comm.q, $1;
        move #/sh.ti3/, st elt.st_name;
        move #/sh.ti3.trace/, st elt.trace_name;
        addq st elt, loc.comm.q, $1;
        move #/sh.ti4/, st elt.st_name;
        move #/sh.ti4.trace/, st elt.trace_name;
        addq st elt, loc.comm.q, $1;
        move #/sh.ti5/, st elt.st_name;
        move #/sh.ti5.trace/, st elt.trace_name;
        addq st elt, loc.comm.q, $1;
    unlock;
endif;
move tasklist, proclist;
sizeq proclist, br.chk;
moved currtask, tempstr;
*move $1, st chk;
while br.chk <> #0;
delq proclist, tcb.name;
cutw tcb.name;
*time curr_tcb.sta.begin;
*move wt.chk, curr_tcb.num;
*add $1, wt.chk, wt.chk;
if (tcb.name <> #/Idle/)
    lock:
        *type nod.name;
        delq loc.comm.q, st elt;
        move tcb.name, st elt.num;
        *type #/--LOCAL-- TASK: /;
        *type tcb.name;
*type */  QUEUE: /;
*typecr st.alt.st_name;
  addq st.alt, temp.q, $1;
  unlock;
  lock;
  delq comm.q, st.alt;
  endwhile;
  delq comm.q, st.alt;
  endwhile;
  move tcb_name, st.alt.norm;
*type */GLOBAL-- TASK: /;
*type */ QUEUE: /;
*typecr st.alt.st_name;
  addq st.alt, comm.q, $1;
  unlock;
  move curpr, cur_tcb.priority;
else;
  move "$0000", cur_tcb.priority;
endif;
  move taskctrl, cur_tcb.pcb;
  move tcb_name, cur_tcb.name;
  move "$/ready", cur_tcb.status;
  move taskctrl, cur_tcb.stk;
  addq cur_tcb, ptr, curpr;
  sizeq procclist, br.chk;
endwhile;
  move temp.q, loc_comm.q;
  move nil_queue, temp.q;
  cxtsw tempstr;
  move "$/rdy_tm/, rdy.str;
  move "$/st_ts/, st_str;
  move "$/run_tm/, run_str;
*move "$/END RESPONSE . . . . . . . . /, end_str;
  bind evt_synch, 40;
  bind evt_lost_synch, 16;
  bind evt_msg_rx, 32;
  bind evt_signals, 33;
  bind evt_clk, 4;
  enable $10;
  printer "$/Kernel Starting.../"
  call run_isr_task;
  printer "$/ERROR - KERNEL TERMINATING/"
  ENDTASK;
  ENDEKERNEL;
B.4.8 Earliest Due Date Application Tasks

| MSG Task |
| STAT rdy_tm, integer; |
| STAT wt_tm, integer; |
| STAT run_tm, integer; |
| VAR byte_cnt, integer; No. of message bytes; |
| VAR msg_pending, integer; |
| VAR msg_out, msg_in.type; |
| VAR msg_in, t_msg_alt; |
| VAR task_strt, integer; |
| VAR task_end, integer; |
| STAT cycle, integer; |
| STAT num_cycles, integer; |
| VAR tot_time, integer; |

**Activity time_upd:**
- begin:
  - task_start;
  - add msg_end, task_strt, task_end;
  - add msg_end, tot_time, tot_time;
  - end;

**Activity init:**
- time task_strt;
- move 0, msg_pending;
- move 0, msg_out;
- end;

**Activity upd_cycle:**
- move 0, msg_in.type;
- add 1, num_cycles, num_cycles;
- div tot_time, num_cycles, cycle;
- time task_end;
- add task_end, task_strt, task_end;
- add task_end, tot_time, tot_time;
- time task_strt;
- end;

**Activity init_cnt:**
- move 0, byte_cnt;
- move 0, msg_rdy;
- end;

**Activity incr_cnt:**
- add 1, byte_cnt, byte_cnt;
- end;

**Activity send_tone:**
- move 0, msg_out.type;
- move msg_due, msg_out, msg_out.prior;
- move msg[msg], msg_out, src;
- end;

**Activity send_start:**
- move 0, msg_out.type;
- move msg_due, msg_out, msg_out.prior;
- move msg[msg], msg_out, src;
- end;

**Activity send_diae:**
- move 0, msg_out.type;
- move msg_due, msg_out, msg_out.prior;
- move msg[msg], msg_out, src;
- end;

**Activity ack_in:**
- time task_strt;
- if (msg_in.type <> 0);
  - move 1, msg_pending;
- endif;
- end;

**Activity clr_msg:**
- move 0, msg_in.type;
- move 0, msg_pending;
- end;

**Activity set_sync:**
- ticks 3200;
- move 0, msg_in.type;
- move 0, msg_pending;
- end;

**Activity time_upd:**
- begin:
  - DO_AKT init;
  - CALLP nilprim;
  - SETSTATE #/clear;
  - LOOP:
    - STATE clear;
      - IF #true;
        - DO_AKT upd_cycle;
        - CALLP nseep;
        - SETSTATE #/meet;
      - ENDIF;
      - ENDSTATE;
    - STATE start;
      - IF (msg_pending = #1);
        - CALLP nilprim;
        - SETSTATE #/hndl_msg/1;
      - ENDIF;
      - IF (msg_rdy = #1);
        - DO_AKT init_cnt;
        - CALLP nilprim;
        - SETSTATE #/rstxx/1;
      - ENDIF;
      - IF #true;
        - DO_AKT time_upd;
        - CALLP recr, 0, 0, #/msg_in/1;
      - SETSTATE #/sigmsg/1;
      - ENDIF;
      - ENDSTATE;
    - STATE sigmsg;
      - IF #true;
        - DO_AKT ack_in;
        - CALLP nilprim;
        - SETSTATE #/start/1;
      - ENDIF;
      - ENDSTATE;
    - STATE rstxx;
      - IF (byte_cnt < #32);
        - DO_AKT incr_cnt;
        - CALLP nilprim;
        - SETSTATE #/rstxx/1;
      - ENDIF;
      - IF #true;
        - CALLP nilprim;
        - SETSTATE #/msg_fin/1;
      - ENDIF;
      - ENDSTATE;
    - STATE msg_fin;
      - IF (msg_type = #1);
        - DO_AKT send_start;
        - CALLP send, #/finish, #/msg_out/1;
        - SETSTATE #/clear/1;
      - ENDIF;
      - IF (msg_type = #2);
        - DO_AKT send_tone;
        - CALLP send, #/tone, #/msg_cst/1;
        - SETSTATE #/clear/1;
      - ENDIF;
      - IF (msg_type = #3);
        - DO_AKT send_diae;
        - CALLP send, #/diage, #/msg_out/1;
        - SETSTATE #/clear/1;
      - ENDIF;
      - ENDSTATE;
    - STATE hndl_msg;
      - IF (msg_in_type = #7);
        - DO_AKT clr_msg;
        - CALLP nseep, #1;
        - SETSTATE #/clear/1;
      - ENDIF;
    - ENDSTATE;
  - ENDSTATE;
- ENDIF;
IF (msg.in.type = $0);
  DO_ACT set_sync;
  CALLP endresp, $2;
ENDIF;
ENDSTATE;
ENDSELECT;
ENDLOOP
ENDTASK;

; SYNCH Task

STAT ryd_tm, integer;
STAT ut_tm, integer;
STAT run_tm, integer;

VAR msg_out, t_msg_ctr;
VAR task_strt, integer;
VAR task_end, integer;
STAT cycle, integer;
STAT num_cycles, integer;
VAR tot_time, integer;

ACTIVITY init;
  *time task_strt;
  move $0, synch_loss;
ENACT;

ACTIVITY upd_cycle;
  *add $1, num_cycles, cycle;
  div tot_time, num_cycles, cycle;
  *time task_end;
  *sub task_end, task_strt, task_end;
  *add task_end, tot_time, tot_time;
  *time task_strt;
ENACT;

ACTIVITY report_synch;
  *time task_strt;
  move $0, synch_loss;
  move l_synch, msg_out.policy;
  move #/synch, msg_out.src;
ENACT;

ACTIVITY change_synch;
  *time task_strt;
  move $8, msg_out.type;
  move ch_synch, msg_out.policy;
  move #/synch, msg_out.src;
ENACT;

ACTIVITY time_upd;
  *time task_end;
  *sub task_end, task_strt, task_end;
  *add task_end, tot_time, tot_time;
ENACT;

BEGIN
  DO_ACT init;
  CALLP nilprim;
  SETSTATE #/clear/;
END;

LOOP;
STATESELECT;
STATE clear;
  IF #true;
    DO_ACT upd_cycle;
    CALLP newresp;
  SETSTATE #/start/;
ENDIF;
ENDSTATE;
STATE start;
  IF #true;
    DO_ACT time_upd;
    CALLP sleep;
  SETSTATE #/chk/;
END;

ENDSTATE;
ENDSTATE;
ENDSTATE;
ENDSTATE;
ENDSELECT;
ENDLOOP
ENDTASK;

; FAULT Task

STAT ryd_tm, integer;
STAT ut_tm, integer;
STAT run_tm, integer;

VAR msg_out, t_msg_ctr;
VAR msg_in, t_msg_ctr;
VAR task_strt, integer;
VAR task_end, integer;
STAT cycle, integer;
STAT num_cycles, integer;
VAR tot_time, integer;

ACTIVITY init;
  *time task_strt;
ENACT;

ACTIVITY upd_cycle;
  *add $1, num_cycles, cycle;
  div tot_time, num_cycles, cycle;
  *time task_end;
  *sub task_end, task_strt, task_end;
  *add task_end, tot_time, tot_time;
  *time task_strt;
ENACT;

ACTIVITY time_upd;
  *time task_end;
ENACT;

ACTIVITY send_report;
  move #7, msg_out.type;
  move msg_in.type, msg_out.policy;
  move #/fault, msg_out.src;
ENACT;

BEGIN
  DO_ACT init;
  CALLP nilprim;
  SETSTATE #/clear/;
END;

ENDSTATE;
STATE clear;
  IF #true;
    DO_ACT upd_cycle;
    CALLP newresp;
  SETSTATE #/start/;
ENDIF;
ENDSTATE;
STATE start;
  IF #true;

ENDSTATE;
ENDSTATE;
ENDSTATE;
ENDSTATE;
ENDSELECT;
ENDLOOP
ENDTASK;
DO_ACT: time_upd;
CALLP rev, #0, #msg_in/1
SETSTATE #/got_msg/
ENDIF;
ENDSTATE;

STATE got_msg:
IF (msg_in.src = #/time/)
DO_ACT: time_mid;
CALLP get_data, msg_in, src, #/time_data/1, #1000;
SETSTATE #/report/1
ENDIF;

IF (msg_in.src = #/stat/)
DO_ACT: time_stat;
CALLP get_data, msg_in, src, #/stat_data/1, #1000;
SETSTATE #/report/1
ENDIF;

IF (true)
DO_ACT: time_st;
CALLP nilprim;
SETSTATE #/report/1
ENDIF;
ENDSTATE;

STATE report:
IF (true)
DO.ACT: send_report;
CALLP send, #/msg/1, #/msg_out/1;
SETSTATE #/clear/1
ENDIF;
ENDSTATE;
ENDSELECT;
ENDLOOP
ENDTASK;

; AUDITS Task

STAT rcy_tm, integer;
STAT wr_tm, integer;
STAT run_tm, integer;

VAR msg_out, tmsg_out;
VAR aud1_cnt, integer;
VAR aud2_cnt, integer;
VAR aud3_cnt, integer;
VAR task_start, integer;
VAR task_end, integer;
VAR cycle, integer;
VAR num_cycles, integer;
VAR tot_time, integer;

ACTIVITY init:
    stime task_start;
ENDACT;

ACTIVITY upd_cycle:
    add #1, num_cycles, num_cycles;
    div tot_time, num_cycles, cycle;
    stime task_end;
    sub task_end, task_start, task_end;
    add task_end, tot_time, tot_time;
    stime task_start;
ENDACT;

ACTIVITY incr_cnt:
    stime task_end;
    subb task_end, task_start, task_end;
    add task_end, tot_time, tot_time;
    add #1, aud1_cnt, aud1_cnt;
    add #1, aud2_cnt, aud2_cnt;
    add #1, aud3_cnt, aud3_cnt;
ENDACT;

ACTIVITY tell_time:
    stime task_start;
    move #22, msg_out.type;
ACTIVITY audit1:
    move #0, aud1_cnt;
    move msg_out, audit1_prior;
    move R10, msg_out.type;
    move #/audit1, msg_out.src;
ENDACT;

ACTIVITY audit2:
    move #0, aud2_cnt;
    move msg_out, audit2_prior;
    move #/audit2, msg_out.src;
ENDACT;

ACTIVITY audit3:
    move #0, aud3_cnt;
    move R10, msg_out.type;
    move msg_out, audit3_prior;
    move #/audit3, msg_out.src;
ENDACT;

BEGIN
DO.ACT: init;
CALLP nilprim;
SETSTATE #/clear/1;
END;

LOOP:

STATESELECT:

STATE clear:
    IF (true)
    DO.ACT: upd_cycle;
    CALLP move, #/move/1
    SETSTATE #/start/1
    END;
    ENDSTATE;

STATE start:
    IF (true)
    DO.ACT: incr_cnt;
    CALLP sleep;
    SETSTATE #/trigger_time/1
    END;
    ENDSTATE;

STATE trigger_time:
    IF (true)
    DO.ACT: tell_time;
    CALLP send, #/time/1, #/msg_out/1
    SETSTATE #/chk_aud1/1
    END;
    ENDSTATE;

STATE chk_aud1:
    IF (true)
    DO.ACT: aud1;
    CALLP send, #/dia/1, #/msg_out/1
    SETSTATE #/chk_aud2/1
    END;
    IF (true) CALLP nilprim;
    SETSTATE #/chk_aud2/1
    END;
    ENDSTATE;

STATE chk_aud2:
    IF (true)
    DO.ACT: aud2;
    CALLP send, #/dia/1, #/msg_out/1
    SETSTATE #/chk_aud3/1
    END;
    IF (true) CALLP nilprim;
    SETSTATE #/chk_aud3/1
    END;
    ENDSTATE;

STATE chk_aud3:
    IF (true)
    DO.ACT: aud3;
    CALLP send, #/dia/1, #/msg_out/1
    END;
    ENDSTATE;

457
### T1MTC Task

STAT rdy_tm, integer;
STAT st_tm, integer;
VAR msg_out, t_msg_elt;
VAR msg_in, t_msg_elt;
VAR result, integer;
VAR fail, integer;
VAR task_strt, integer;
VAR task_end, integer;
STATE cycle, integer;
STATE num_cycles, integer;
VAR tot_time, integer;

**ACTIVITY init:**
`time task_strt;`

**ACTIVITY upd_cycle:**
`*add $1, num_cycles, num_cycles;`
`*div tot_time, num_cycles, cycle;`
`*time task_end;`
`*sub task_end, task_strt, task_end;`
`*add task_end, tot_time, tot_time;`
`*time task_strt;`

**ACTIVITY time_upd:**
`*time task_end;`
`*add task_end, task_strt, task_end;`
`*add task_end, tot_time, tot_time;`

**ACTIVITY time_st:**
`*time task_strt;`

**ACTIVITY do_scan:**
`*time task_strt;`
`*add $0, fail;`
`move $1, t_data;`
`tick $100;`
`rand $10, result;`
`if (result == $0);`
`move $1, fail;`
`endif;`

**ACTIVITY report_fail:**
`move $10, msg_out.type;`
`move $5, msg_out.prior;`
`move $/tmc/, msg_out.src;`

**BEGIN**
`DO_ACT init;`
`CALLP nilprim;`
`STATE #/clear/;`

**LOOP:**

**STATESELECT:**

**STATE clear:**
`IF #true;`
`DO_ACT upd_cycle;`
`CALLP neuresp;`
`STATE #/start/;`

**ENDIF:**

**STATE start:**
`IF #true;`
`DO_ACT time_upd;`
`CALLP recv, 0, #/msg_in/;`
`STATE #/got_msg/;`

**ENDIF:**

**STATE got_msg:**
`IF (msg_in.type == $23);`
`DO_ACT do_scan;`
`CALLP nilprim;`
`STATE #/chk_results/;`

**ENDIF:**

**STATE chk_results:**
`IF #true;`
`DO_ACT report_fail;`
`CALLP send, #/fault/, #/msg_out/;`
`STATE #/response/;`

**ENDIF:**

**STATE response:**
`IF #false;`
`CALLP nilprim;`
`STATE #/clear/;`

**ENDIF:**

**ENDSTATE:**

### DIAGS Task

STAT rdy_tm, integer;
STAT st_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_elt;
VAR msg_in, t_msg_elt;
VAR dest, string;
VAR dg_num, integer;
VAR task_strt, integer;
VAR task_end, integer;
STATE cycle, integer;
STATE num_cycles, integer;
VAR tot_time, integer;

**ACTIVITY init:**
`time task_strt;`

**ACTIVITY upd_cycle:**
`*add $1, num_cycles, num_cycles;`
`*div tot_time, num_cycles, cycle;`
`*time task_end;`
`*sub task_end, task_strt, task_end;`
`*add task_end, tot_time, tot_time;`
`*time task_strt;`

**ENDSTATE:**

---

458
```c
sub  tse_end, tse_start, tse_end;
add  tse_end, tot_time, tot_time;
ENDACT;

ACTIVITY time_st;
etime tse_start;
ENDACT;

ACTIVITY chnl_req;
etime tse_start;
mov $12, msg_out.type;
mov msg_in.prev, msg_out.prev;
mov #msg_out/src, msg_out.xrc;
ENDACT;

ACTIVITY choose_diag;
etime tse_start;
rand $4, dg_num;
mov $21, msg_out.type;
mov msg_in.prev, msg_out.prev;
mov #diag/s, msg_out.xrc;
if (dg_num = #0)
   mov #tone_diag/, dest;
else
   if (dg_num = #1)
      mov #chldiag/, dest;
   else
      if (dg_num = #2)
         mov #msgdiag/, dest;
      else
         mov #sidia/g, dest;
   endif;
endif;
endif;
ENDACT;

BEGIN
DO.ACT  init;
CALLP nilprim;
SETSTATE #/clear/;

LOOP:
STATESSELECT;

STATE clear;
if #true;
   DO.ACT  upd_cycle;
   CALLP  newresp;
   SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
if #true;
   DO.ACT  time_upd;
   CALLP  recv, #3, #/msg_in/;
   SETSTATE #/get_msg/;
ENDIF;
ENDSTATE;

STATE got_msg;
if #true;
   DO.ACT  chnl_req;
   CALLP  send, #/chnl/, #/msg_out/;
   SETSTATE #/wt.ack1/;
ENDIF;
ENDSTATE;

STATE wt.ack1;
if #true;
   DO.ACT  time_upd;
   CALLP  recv, #14, #/msg_in/;
   SETSTATE #/got_chnl/;
ENDIF;
ENDSTATE;

STATE got_chnl;
if #true;
   DO.ACT  chnl_req;
   CALLP  send, #/chnl/, #/msg_out/;
   SETSTATE #/wt.ack2/;
ENDIF;
ENDSTATE;

STATE wt.ack2;
if #true;
   DO.ACT  time_upd;
   CALLP  recv, #14, #/msg_in/;
   SETSTATE #/pick_diag/;
ENDIF;
ENDSTATE;

STATE pick_diag;
if #true;
   DO.ACT  choose_diag;
   CALLP  send, dest, #/msg_out/;
   SETSTATE #/clear/;
ENDIF;
ENDSTATE;

ENDSELECT;
ENDLOOP
ENDTASK;
```
; TIDIG Task

STAT rdy_tm, integer;
STAT wt_tm, integer;
STAT run_tm, integer;

VAR mag_out, t_meg_elt;
VAR mag_in, t_meg_elt;
VAR result, integer;
VAR fail, integer;
VAR tk_strt, integer;
VAR tk_end, integer;
STAT cycle, integer;
STAT num_cycles, integer;
VAR tot_time, integer;

ACTIVITY init;
  +time tk_strt;
ENDACT;

ACTIVITY upd_cycle;
  +add $1, num_cycles, num_cycles;
  +div tot_time, num_cycles, cycle;
  +time tk_end;
  +sub tk_end, tk_strt, tk_end;
  +add tk_end, tot_time, tot_time;
  +time tk_strt;
ENDACT;

ACTIVITY time_upd;
  +time tk_end;
  +sub tk_end, tk_strt, tk_end;
  +add tk_end, tot_time, tot_time;
ENDACT;

ACTIVITY stop_mtc;
  +time tk_strt;
  move $00, mag_out;
  move $00, mag_out.prior;
  move $00, mag_out.prior;
  move $00, mag_out.prior;
ENDACT;

ACTIVITY do_diag;
  move $00, fail;
  tick $000;
  rand $10, result;
  if (result = $0); move $1, fail;
ENDACT;

ACTIVITY report_fail;
  move $10, mag_out.prior;
  move mag_in.prior, mag_out.prior;
ENDACT;

BEGIN
  DO_ACT init;
  CALLP nilprim;
  SETSTATE #/clear/;
ENDACT;

ENDSTATE;

STATE got_msg;
  IF true;
    DO_ACT stop_mtc;
    CALLP send, #/signal/, #/msg_out/;
    SETSTATE #/diagnose/;
  ENDIF;
ENDSTATE;

STATE diagnose;
  IF true;
    DO_ACT do_diag;
    CALLP nilprim;
    SETSTATE #/chk_results/;
  ENDIF;
ENDSTATE;

STATE chk_results;
  IF (fail = $1);
    DO_ACT report_fail;
    CALLP send, #/result/, #/msg_out/;
    SETSTATE #/resom/;
  ENDIF;
ENDSTATE;

STATE resom;
  IF (fail = $0);
    CALLP send, #/resom/;
    SETSTATE #/clean/;
  ENDIF;
ENDSTATE;

STATE end_res;
  IF true;
    CALLP nilprim;
    SETSTATE #/clean/;
  ENDIF;
ENDSTATE;

ENDSTATE;

ENDSELECT;
ENDLOOP ENDTASK;

; MSGDIAG Task

STAT rdy_tm, integer;
STAT wt_tm, integer;
STAT run_tm, integer;

VAR mag_out, t_meg_elt;
VAR mag_in, t_meg_elt;
VAR result, integer;
VAR fail, integer;
VAR tk_strt, integer;
VAR tk_end, integer;
STAT cycle, integer;
STAT num_cycles, integer;
VAR tot_time, integer;

ACTIVITY init;
  +time tk_strt;
ENDACT;

ACTIVITY upd_cycle;
  +add $1, num_cycles, num_cycles;
  +div tot_time, num_cycles, cycle;
  +time tk_end;
  +sub tk_end, tk_strt, tk_end;
  +add tk_end, tot_time, tot_time;
  +time tk_strt;
ENDACT;

ACTIVITY time_upd;
  +time tk_end;
  +sub tk_end, tk_strt, tk_end;
  +add tk_end, tot_time, tot_time;
ENDACT;

ACTIVITY stop_integ,
ENDSELECT;
ENDLOOP
ENDTASK;

<table>
<thead>
<tr>
<th>CHM_DIAG Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>STAT rdg_tm, integer;</td>
</tr>
<tr>
<td>STAT st_tm, integer;</td>
</tr>
<tr>
<td>STAT run_tm, integer;</td>
</tr>
</tbody>
</table>

VAR msg_out, t_msg_alt; |
VAR msg_in, t_msg_alt; |
VAR res, integer; |
VAR fail, integer; |
VAR tskstrt, integer; |
VAR tsk_end, integer; |
STAT cycle, integer; |
STAT num_cycles, integer; |
VAR tot_time, integer; |

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
LOOP;
STATESELECT;

STATE clear;
IF #true;
DO_ACT do_diag;
CALLP msg_out, msg_out prio;
SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
IF #true;
DO_ACT time_upd;
CALLP recv, NO, #/msg_in/;
SETSTATE #/got_msg/;
ENDIF;
ENDSTATE;

STATE got_msg;
IF #true;
DO_ACT stop_integ;
CALLP send, #/signal/, #/msg_out/;
SETSTATE #/diagnose/;
ENDIF;
ENDSTATE;

STATE diagnose;
IF #true;
DO_ACT do_diag;
CALLP nilprim;
SETSTATE #/chk_results/;
ENDIF;
ENDSTATE;

STATE chk_results;
IF (fail = #1);
DO_ACT report_fail;
CALLP send, #/fault/, #/msg_out/;
SETSTATE #/response/;
ENDIF;
ENDIF;
CALLP nilprim;
SETSTATE #/response/;
ENDIF;
ENDSTATE;

STATE response;
IF (fail = #0);
CALLP endresp, NO;
SETSTATE #/clear/;
ENDIF;
ENDIF;
CALLP nilprim;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

BEGIN
DO_ACT init;
CALLP nilprim;
SETSTATE #/clear/;
LOOP;
STATESELECT;

STATE clear;
IF #true;
DO_ACT upd_cycle;
CALLP neuresp;
ENDIF;
ENDIF:
CALLP nilprim;
SETSTATE #/start/;
ENDSTATE;

STATE start;
IF @true;
DO_ACT time_upd;
CALLF rxcv, @0, #/msg_in/;
SETSTATE #/got_msg/;
ENDIF;
ENDSTATE;

STATE got_msg;
IF @true;
DO_ACT connect_chnl;
CALLF send, #/connect/, #/msg_out/;
SETSTATE #/signal/;
ENDIF;
ENDSTATE;

STATE signal;
IF @true;
DO_ACT stop_integ;
CALLF send, #/signal/, #/msg_out/;
SETSTATE #/diagnose/;
ENDIF;
ENDSTATE;

STATE diagnose;
IF @true;
DO_ACT do_diag;
CALLF nilprim;
SETSTATE #/chk_results/;
ENDIF;
ENDSTATE;

STATE chk_results;
IF (fail = @1); DO_ACT report_fail;
CALLF send, #/fail/,
SETSTATE #/response/;
ENDIF;
IF @true;
CALLF nilprim;
SETSTATE #/response/;
ENDIF;
ENDSTATE;

STATE response;
IF (fail = @0); DO_ACT endresp, @0;
CALLF #/clear/;
SETSTATE #/nilprim/;
ENDIF;
IF @true;
CALLF nilprim;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

ENDSELECT;
ENDLOOP
ENDTASK;

BEGIN
DO_ACT init;
CALLF nilprim;
SETSTATE #/clear/;
LOOP
STATE SELECT;

STATE clear;
IF @true;
DO_ACT upd_cycle;
CALLF numresp;
SETSTATE #/start/;
ENDIF;
ENDSTATE;

STATE start;
IF @true;
DO_ACT time_upd;
CALLF rxcv, @1, #/msg_in/;
SETSTATE #/got_msg/;
ENDIF;
ENDSTATE;

STATE got_msg;
IF @true;
DO_ACT chnl_req;
CALLF send, #/chnl/,
SETSTATE #/got_chnl/;
ENDIF;
ENDSTATE;

STATE wt_ack;
IF @true;
DO_ACT time_upd;
CALLF rxcv, @4, #/msg_in/;
SETSTATE #/got_msg/;
ENDIF;
ENDSTATE;

STATE got_chnl;
IF @true;
DO_ACT conn_req;
CALLF send, #/connect/, #/msg_out/;
SETSTATE #/got_conn/;
ENDIF;
ENDSTATE;

ACTIVITY init;
@end time tak_strt;

ACTIVITY upd_cycle;
@end add @1, num_cycles, num_cycles;
@end div tot_time, num_cycles, cycle;
@end time tak_end;
@end sub tak_end, tak_strt, tak_end;
@end add tak_end, tot_time, tot_time;
@end time tak_strt;

ACTIVITY time_upd;
@end time tak_end;
@end sub tak_end, tak_strt, tak_end;
@end add tak_end, tot_time, tot_time;

ACTIVITY chnl_req;
@end time tak_strt;
move @12, msg_out.type;
move msg_in.prior, msg_out.prior;
move #/cin/,

ACTIVITY conn_req;
@end time tak_strt;
move @11, msg_out.type;
move msg_in.prior, msg_out.prior;
move #/cin/,

ACTIVITY stat_req;
@end move @10, msg_out.type;
move msg_in.prior, msg_out.prior;
move #/cin/,

ACTIVITY stat_clear;
@end move @9, msg_out.type;
move msg_in.prior, msg_out.prior;
move #/cin/,

ACTIVITY stat_wt_ack;
@end move @8, msg_out.type;
move msg_in.prior, msg_out.prior;
move #/cin/,
BEGIN

STATE got_comm;
  IF true;
  DO_ACT stat_req;
  CALLP send, #stat, #msg_out/1;
  SETSTATE #/clear/1;
  ENDIF;
  ENDSTATE;
ENDSELECT;
ENDLOOP
ENDTASK;

| | CHNLs Task |
| | |
STAT rdy_tm, integer;
STAT wt_tm, integer;
STAT num_tm, integer;
VAR msg_out, t_msg_end;
VAR msg_in, t_msg_end;
VAR num_allocated, integer;
VAR tak_strt, integer;
VAR tak_end, integer;
STAT cycle, integer;
STAT num_cycles, integer;
VAR tot_time, integer;
STAT mtchc, integer;
STAT cph, integer;
STAT over_limit, integer;

ACTIVITY initial;
  *time tak_strt;
ENDACT;

ACTIVITY upd_cycle;
  *time tak_strt;
  *add #1, num_cycles, num_cycles;
  *div tot_time, num_cycles, cycle;
  *time tak_end;
  *sub tak_end, tak_strt, tak_end;
  *add tak_end, tot_time, tot_time;
  *time tak_strt;
ENDACT;

ACTIVITY time_upd;
  *time tak_end;
  *sub tak_end, tak_strt, tak_end;
  *add tak_end, tot_time, tot_time;
ENDACT;

ACTIVITY get_chnl;
  *time tak_strt;
  *add #1, num_allocated, num_allocated;
  *if (msg_in.src = #/diag/1);
  *else;
  *endif;
  move #14, msg_out_type;
  move msg_in.prior, msg.out.prior;
  move #/chnl/, msg.out.src;
ENDACT;

ACTIVITY orfly;
  *time tak_end;
  *sub tak_end, tak_strt, tak_end;
  *add tak_end, tot_time, tot_time;
  *add #1, over_limit;
ENDACT;

ACTIVITY free_chnl;
  *time tak_strt;
  *add num_allocated, #1, num_allocated;
  *if (num_allocated < #0);
  move #0, num_allocated;
  endif;
ENDACT;

ENDACT

DO_ACT initial;
CALLP nilprim;
SETSTATE #/clear/1;

LOOP;

STATESELECT;

STATE clear;
  IF true;
  DO_ACT upd_cycle;
  CALLP newresp;
  SETSTATE #/start/1;
  ENDIF;
  ENDSTATE;
ENDSELECT;
ENDLOOP
ENDTASK;

STATE start;
  IF (num_allocated < #16);
  DO_ACT time_upd;
  CALLP recv, #0, #/msg_in/1;
  SETSTATE #/got_msg/1;
  ENDIF;
  IF true;
  DO_ACT orfly;
  CALLP recv, #13, #/msg_in/1;
  SETSTATE #/free_req/1;
  ENDIF;
  ENDSTATE;
ENDSELECT;
ENDLOOP
ENDTASK;

| | STAT Task |
| | |
STAT rdy_tm, integer;
STAT wt_tm, integer;
STAT num_tm, integer;
VAR msg_out, t_msg_end;
VAR msg_in, t_msg_end;
VAR cycle, integer;
VAR num_cycles, integer;
VAR tot_time, integer;

ACTIVITY init;
  *time tak_strt;
ENDACT;

ACTIVITY upd_cycle;
  *add #1, num_cycles, num_cycles;
  *div tot_time, num_cycles, cycle;
  *time tak_end;
  *sub tak_end, tak_strt, tak_end;
  *add tak_end, tot_time, tot_time;
  *time tak_strt;
ENDACT;

463
SETSTATE  #/rel_chal/;
ENDIF;
ENDSTATE;
STATE rel_chal;
IF #true;
DO_ACT;
chal_req;
CALLP send, #/chall/ , #/msg_out/;
SETSTATE  #/call_fin/;
ENDIF;
ENDSTATE;
STATE call_fin;
IF #true;
DO_ACT;
stat_req;
CALLP send, #/stat/ , #/msg_out/;
SETSTATE  #/clear/;
ENDIF;
ENDSTATE;
ENDSELECT;
ENDLOOP
ENDTASK;

; ; CONNECT Task
 ;
STAT rdy_tm, integer;
STAT ut_tm, integer;
STAT run_tm, integer;
VAR msg_in, t_msg_out;
VAR tas_strt, integer;
VAR tas_end, integer;
STAT cycle, integer;
STAT num_cycles, integer;
VAR tot_time, integer;

ACTIVITY init;
  *time tas_strt;
ENDACT;

ACTIVITY upd_cycle;
  *add $1, num_cycles, num_cycles;
  *div tot_time, num_cycles, cycle;
  *time tas_end;
  *sub tas_end, tas_strt, tas_end;
  *add tas_end, tot_time, tot_time;
  *time tas_strt;
ENDACT;

ACTIVITY time_upd;
  *time tas_end;
  *add tas_end, tas_strt, tas_end;
  *add tas_end, tot_time, tot_time;
ENDACT;

ACTIVITY mak_conn;
  *time tas_strt;
  *tick $00;
ENDACT;

BEGIN
  DO_ACT init;
  CALLP nilprim;
  SETSTATE  #/clear/;
LOOP:
  STATE SELECT;
  STATE clear;
  IF #true;
    DO_ACT;
    CALLP
    reqレス;
    SETSTATE  #/start/;
  ENDIF;
  ENDSTATE;
  STATE start;
  IF #true;
    DO_ACT;
    CALLP
    time_upd;
    recv, $0, #/msg_in/;
  ENDIF;
  ENDSTATE;
  ENDSELECT;
ENDLOOP
ENDTASK;

; ; SIGNAL Task
 ;
STAT rdy_tm, integer;
STAT ut_tm, integer;
STAT run_tm, integer;
VAR msg_out, t_msg_out;
VAR msg_in, t_msg_out;
VAR msg_pend, integer;
VAR tas_strt, integer;
VAR tas_end, integer;
STAT cycle, integer;
STAT num_cycles, integer;
VAR tot_time, integer;

ACTIVITY init;
  *time tas_strt;
  move $0, msg_pend;
  move $0, msg_in.type;
ENDACT;

ACTIVITY upd_cycle;
  move $0, msg_in.type;
  *add $1, num_cycles, num_cycles;
  *div tot_time, num_cycles, cycle;
  *time tas_end;
  *sub tas_end, tas_strt, tas_end;
  *add tas_end, tot_time, tot_time;
  *time tas_strt;
ENDACT;

ACTIVITY time_upd;
  *time tas_end;
  *add tas_end, tas_strt, tas_end;
  *add tas_end, tot_time, tot_time;
ENDACT;

ACTIVITY mak_conn;
  *time tas_strt;
  *tick $00;
ENDACT;

BEGIN
  DO_ACT init;
  CALLP nilprim;
  SETSTATE  #/clear/;
LOOP:
  STATE SELECT;
  STATE clear;
  IF #true;
    DO_ACT;
    CALLP
    reqレス;
    SETSTATE  #/start/;
  ENDIF;
  ENDSTATE;
  STATE start;
  IF #true;
    DO_ACT;
    CALLP
    time_upd;
    recv, $0, #/msg_in/;
  ENDIF;
  ENDSTATE;
  ENDSELECT;
ENDLOOP
ENDTASK;
move $3, msg_out.type;
move sig.due, msg_out.prior;
move $/signal/, msg_out.src;
ENDACT;

ACTIVITY ack.in;
etime tsk.stat;
if (msg.in.type <> $0);
move $1, msg.pending;
endif;
ENDACT;

ACTIVITY clr.upd.integ;
move $0, msg.in.type;
ticks #76;
move $0, msg.pending;
ENDACT;

BEGIN
DO.ACT init;
CALLP nilprim;
SETSTATE $/clear/;
ENDACT;

LOOP

STATE SELECT;

STATE clear;
if $true;
DO.ACT upd.cycle;
CALLP newsec;
SETSTATE $/start/;
ENDIF;
ENDSTATE;

STATE start;
if (msg.pending = $1);
CALLP nilprim;
SETSTATE $/hdl_mag/;
ENDIF;
ENDSTATE;

STATE sigmag;
if $true;
DO.ACT ack.in;
CALLP nilprim;
SETSTATE $/start/;
ENDIF;
ENDSTATE;

STATE ex;
if (sig.type = #4);
DO.ACT upd.integ;
CALLP nilprim;
SETSTATE $/clear/;
ENDIF;

ENDSTATE;

STATE diag;
if $true;
DO.ACT send.diag;
CALLP send, $/diag/, $/msg.out/;
SETSTATE $/clear/;
ENDIF;
ENDSTATE;

STATE hndl.mag;
if $true;
DO.ACT clr.upd.integ;
CALLP nilprim;
SETSTATE $/clear/;
ENDIF;
ENDSTATE;

ENDSELECT;
ENDLOOP
ENDTASK;

ENDSTATE;

ENDSTATE;

ENDSTATE;

BEGIN
DO.ACT init;
CALLP nilprim;
SETSTATE $/clear/;
ENDACT;

ACTIVITY time tsk.stat;
if $true;
DO.ACT upd.cycle;
add $1, num.cycles, num.cycles;
div num.cycles, num.cycles, cycle;
time tsk.end;
sub tsk.end, tsk_stat, tsk_end;
add tsk.end, tot_time, tot_time;
time tsk_stat;
ENDIF;

ACTIVITY time_upd;
if $true;
DO.ACT upd_time;
move $1, msg.out.data;
ENDIF;

ACTIVITY select_tone;
time tsk_stat;
ticks #80;
if (msg.in.src = $/mag/);
move $1, msg.out.data;
#else;
move $0, msg.out.data;
endif;
move $11, msg.out.type;
move msg.in.prior, msg.out.prior;
move $/tone/., msg.out.src;
ENDIF;

BEGIN
DO.ACT init;
CALLP nilprim;
SETSTATE $/clear/;
ENDACT;

STATE SELECT;

STATE clear;
if $true;
DO.ACT upd_cycle;
CALLP newsec;
SETSTATE $/start/;
ENDIF;
ENDSTATE;

STATE start;
if $true;
DO.ACT time_upd;
ENDIF;
ENDSTATE;

466
CALLP recv, 0, #msg_in/
STATE get_mag:
IF #true:
DO_ACT:
select_tone;
CALLP send, #connect, #msg_out/;
STATE_set:
#clear/;
ENDIF;
ENDSTATE;
ENDLOOP
ENDTASK:

STATE clear:
IF #true:
DO_ACT:
update;
CALLP wait, #sleep, #msg_out/;
STATE_set:
#start/;
ENDIF;
ENDSTATE;

STATE start:
IF #true:
DO_ACT:
time_upd;
CALLP recv, 0, #msg_in/;
STATE_set:
#got_mag/;
ENDIF;
ENDSTATE;

STATE get_mag:
IF #true:
DO_ACT:
select_tone;
CALLP send, #tune, #msg_out/;
STATE_set:
#diagnose/;
ENDIF;
ENDSTATE;

STATE diagnose:
IF #true:
DO_ACT:
diag;
CALLP nilprim;
STATE_set:
#chk_results/;
ENDIF;
ENDSTATE;

STATE chk_results:
IF (fail = 0):
DO_ACT:
report_fail;
CALLP send, #fault, #msg_out/;
STATE_set:
#response/;
ENDIF;
ENDSTATE;

STATE response:
IF (fail = 0):
CALLP endresp, 0;
STATE_set:
#clear/;
ENDIF;
ENDSTATE;

BEGIN
DO_ACT:
init;
CALLP nilprim;
STATE_set:
#clear/;
ENDSTATE;
LOOP;

ENDSELECT:
B.4.9 Least Slack Time Scheduling Kernel

This section contains the kernel software used to implement the "least slack time" scheduling algorithm. The node file used in this system is also provided as it contains some new type definitions.

```
NODE FILE

| t_trace_elt.act Values: | #1 - Mega Sent  |
|                         | #2 - Mega Accepted |
|                         | #3 - Shared Data Write |
|                         | #4 - Shared Data Read |
|                         | #5 - Response Out |
|                         | #6 - Event Trigger |
|                         | #7 - Start Read Delimiter |
|                         | #8 - End Read Delimiter |
|                         | #9 - Start Msg Delimiter |
|                         | #10 - End Msg Delimiter |

TYPE t_trace_elt = node_name:string *
  task_name:string *
  act :integer *
  time :integer *
  misc1 :integer *
  misc2 :integer *

TYPE t_msg_elt = data :integer *
  type :integer *
  dau :integer *
  elapse :integer *
  length :integer *
  arc :string *
  two_sane :integer; No. of associated trace elts

VAR nod_name, string; value must be initialized in kernel task
VAR msg_rdy, integer; used for external messages by MSG
VAR msgtype, integer; used for external messages by MSG
VAR synch_loss, integer; used by SYNCH
VAR sig_rdy, integer; used for signalling messages by SIGNAL
VAR sigtype, integer; used for signalling messages by SIGNAL
VAR sce_owhd, integer;

EVENT clk1;
KERNEL base;
TASK msg, 10;
TASK idle, 1;
ENDMODE;
```
KERNEL CODE FILE

This kernel contains the following primitives:
  nonblocking send
  blocking selective receive
  sleep
  get data from another task
  nilprim

The following external message types are supported:
  1. - start call setup
  2. - provide tone
  3. - run diagnostic
  4. - integrity ok
  5. - integrity failure
  6. - call takedown

The following internal message types are supported:
  7. - START - request call setup
  8. - TONES - request for tone
  9. - DIAG - request single random diagnostic
  10. -
  11. -
  12. - MSG - report fault
  13. - MSG - perform and report synch operation
  14. - FAULT - synch lost
  15. - FAULT - general fault report
  16. - CONNECT - perform connect/disconnect
  17. - CHLS - allocate channel
  18. - CHLS - free channel
  19. - all tasks - channel granted
  20. - STAT - call setup complete
  21. - SIGNAL - start integrity checking
  22. - STAT - integrity failure report
  23. - END - caller hung up
  24. - STAT - call finished
  25. - STAT - stop integrity checking
  26. - all diags - run diagnostic
  27. - TIMT - stop carrier checking
  28. - TIMT - monitor carriers

The following response types are supported:
  1. - MSG - fault report sent to switch center
  2. - MSG - synch source changed
  3. - CONNECT - switch center tone request connected
  4. - STAT - call finished
  5. - STAT - call started
  6. - diags - diagnostic completed without failure
  7. - TIMT - carrier scan completed without failure

KERNEL baset:

TASK kern_task, 50;

TYPE t_tcb_status = rd_time: integer *;
  run_time: integer *;
  ut_time: integer *;
  begin: integer *;
  finish: integer *;
  resp_q: queue; of t_traces_elt

TYPE t_shared_status = data_trace: queue; of t_traces_elt

TYPE t_tcb = name: string *; task name
  num: integer *;
  ddate: integer *; due data
  elapse: integer *; elapsed response time
  status: string *; status
  priority: integer *; priority
  pcount: integer *; program counter
  elb_str: string *; elb at str
  stk: stack *; stack
  sta: t_tcb_status; task control block

TYPE t_sh_data = t_br: string *; task name
  num_msg: integer *; number of queued messages
  msg: queue *; message queue (t_msg_elt)
  tsc: queue *; trace queue (t_trace_elt)
TYPE t_msg_struct = task_num : string *;
    task name
waiting : integer *;
    flag set if blocked for msg
msg_q : queue *;
    message queue (t_msg_elt)
traces : queue *;
    trace queue (t_trace_elt)
w_type : integer *;
    type of msg blocked on
block : t_tcb;
    tcb wait area

TYPE t_id_struct = nom : string *;
    of t_id_struct (one elt per task);
trc_name : string *;
    task name : string

SHARED comm_q, queue; of t_id_struct (one elt per task);
SHARED th_t1, t_sh_data;
SHARED th_t2, t_sh_data;
SHARED th_t3, t_sh_data;
SHARED sh_t4, t_sh_data;
SHARED sh_t5, t_sh_data;
SHARED sh_t6, t_sh_data;
SHARED sh_t7, t_sh_data;
SHARED sh_t8, t_sh_data;
SHARED sh_t9, t_sh_data;
SHARED sh_t10, t_sh_data;
SHARED sh_t11, t_sh_data;
SHARED sh_t12, t_sh_data;
SHARED sh_t13, t_sh_data;
SHARED sh_t14, t_sh_data;
SHARED sh_t15, t_sh_data;
SHARED sh_t16, t_sh_data;
SHARED sh_t17, t_sh_data;
SHARED ti_data, integer;
SHARED stat_data, integer;
SHARED num_qed, integer;

; Message queues
VAR loc_comm_q, queue; of t_id_struct (one elt per local task)
VAR at_elt, t_id_struct;
VAR l_t1, t_msg_struct;
VAR l_t2, t_msg_struct;
VAR l_t3, t_msg_struct;
VAR l_t4, t_msg_struct;
VAR l_t5, t_msg_struct;
VAR l_t6, t_msg_struct;
VAR l_t7, t_msg_struct;
VAR l_t8, t_msg_struct;
VAR l_t9, t_msg_struct;
VAR l_t10, t_msg_struct;
VAR l_t11, t_msg_struct;
VAR l_t12, t_msg_struct;
VAR l_t13, t_msg_struct;
VAR l_t14, t_msg_struct;
VAR l_t15, t_msg_struct;
VAR l_t16, t_msg_struct;
VAR local, string;
VAR shared, string;
VAR loc_trc, string;
VAR sh_trc, string;

STAT sends, integer;
STAT os_ovhd, integer;
STAT switches, integer;
STAT preempts, integer;
STAT evt_0, integer;
STAT evt_1, integer;
STAT evt_2, integer;
STAT evt_3, integer;
STAT evt_4, integer;
STAT evt_5, integer;
VAR start_os, integer;
VAR end_os, integer;
VAR elapsed, integer;
VAR sc_start, integer;
VAR sc_end, integer;
VAR rdy_str, string;
VAR wt_str, string;
VAR run_str, string;
VAR end_str, string;
VAR wait_q, queue;
VAR ut_chk, integer;
VAR msg_str, string;
VAR dest_str, string;
VAR temp_tcb, t_tcb;
VAR temp_tcb, t_tcb;
VAR ret_pc, integer;
VAR tmp_stk, stack;
VAR nm_name, integer;
VAR found, integer;
VAR msg_type, integer;
VAR data_size, integer;
VAR nil_queue, queue;
VAR nil_string, string;
VAR msg_el, t_msg_el;
VAR n_num, integer;
VAR temp2_q, queue;
VAR mtype, integer;
VAR q_num, integer;
VAR done, integer;
VAR ch_sync, integer;
VAR l_sync, integer;
VAR msgفارق, integer;
VAR sigفارق, integer;
VAR msg褌, integer;
VAR sig褌, integer;
VAR msg褌, integer;
VAR l_sync, integer;
VAR ch_sync, integer;
VAR diag褌, integer;
VAR sim, integer;
VAR sig褌, integer;
VAR try_tcb, t_tcb;
VAR best_tcb, t_tcb;
VAR slack1, integer;
VAR slack2, integer;
VAR cur_time, integer;

; Global Kernel Variables
VAR curr_tcb, t_tcb ; current task control block
VAR proc_list, queue ; copy of task name list
VAR br.chk, integer; ; branch condition check
VAR tempstr, string; ; temporary string value
VAR tcb_name, string; ; current task name
VAR ror, queue; ; ready to run queue

; Variables for get_local and get_sh procedures
VAR g_q_size, integer;
VAR g_found, integer;
VAR g_get_el, t_id_struct;

; Variables for response_update procedure
VAR action, integer; ; action requested in response_update
VAR prio, integer; ; send priority
VAR exec_entry, t_trace_elt; ; new response trace element
VAR msg褌_name, string; ; name of msg褌 for response trace
VAR data褌_name, string; ; name of msg褌 queue
VAR data褌, t_trace_elt; ; msg or data trace delimiter
VAR q_size, integer; ; size of trace queue
VAR data褌name, string; ; name of shared data variable
VAR temp褌, queue; ; data response queue copy
VAR resp_time, integer; ; elapsed response time

; VARIABLE declarations for procedure RUN_RTR_TASK
VAR error, boolean; ; Set if no ready processes

; VARIABLE declarations for procedure SCRRD_RTR_TASK
VAR rtr_tcb, t_tcb; ; PCB to be scheduled

; PROCEDURE response_update (Gname: string;
; action: integer)
; This procedure updates the response trace queues
PROC response_update:
  *pop task_thk, action;
  *move nod_name, trace_elt.node_name;
  *move curr_tcb, trace_elt.task_name;

471
* time  trace_elt.time;
  *move  action, trace_elt.act;

  \*if (action = #1); SEND MSG
  \*pop  taskstk, prc;
  \*pop  taskstk, msg_elt_name;
  \*addq  trace_elt, curr_tcb.sts.resp_q, #1;
  \*move  curr_tcb.sts.resp_q, temp_q;
  \*sizeq  temp_q, q.size;
  \*move  q.size, msg_elt_name.trc.size;
  \*while (q.size > #0);
    \*delq  temp_q, trace_elt;
    \*addq  trace_elt, @dest_elt_name, prc;
    \*sizeq  temp_q, q.size;
  \*endwhile;
  \*endif;

  \*if (action = #2); RCV MSG
  \*pop  taskstk, dest_elt_name;
  \*pop  taskstk, msg_elt_name;
  \*move  trace_elt, delim_elt;
  \*addq  trace_elt, curr_tcb.sts.resp_q, #1;
  \*move  $9, delim_elt.act;
  \*addq  delim_elt, curr_tcb.sts.resp_q, #1;
  \*move  msg_elt_name.trc.size, q.size;
  \*while (q.size > #0);
    \*delq  @dest_elt_name, trace_elt;
    \*addq  trace_elt, curr_tcb.sts.resp_q, #1;
    \*sub  q.size, #1, q.size;
  \*endwhile;

  \*move  #10, delim_elt.act;
  \*addq  delim_elt, curr_tcb.sts.resp_q, #1;
  \*endif;

  \*if (action = #3); DATA WRITE
  \*pop  taskstk, data_name;
  \*addq  trace_elt, curr_tcb.sts.resp_q, #1;
  \*move  curr_tcb.sts.resp_q, @data_name.stats.trace;
  \*endif;

  \*if (action = #4); DATA READ
  \*pop  taskstk, data_name;
  \*move  trace_elt, delim_elt;
  \*addq  trace_elt, curr_tcb.sts.resp_q, #1;
  \*move  $7, delim_elt.act;
  \*addq  delim_elt, curr_tcb.sts.resp_q, #1;
  \*move  @data_name.stats.data_trace, temp_q;
  \*sizeq  temp_q, q.size;
  \*while (q.size > #0);
    \*delq  temp_q, trace_elt;
    \*addq  trace_elt, curr_tcb.sts.resp_q, #1;
    \*sizeq  temp_q, q.size;
  \*endwhile;

  \*move  #8, delim_elt.act;
  \*addq  delim_elt, curr_tcb.sts.resp_q, #1;
  \*endif;

  \*if (action = #5); RESPONSE OUT
  \*pop  taskstk, trace_elt.misc1;
  \*pop  taskstk, resp_time;
  \*addq  trace_elt, curr_tcb.sts.resp_q, #1;
  \*sizeq  curr_tcb.sts.resp_q, q.size;
  \*while (q.size > #0);
    \*delq  curr_tcb.sts.resp_q, trace_elt;
    \*type  \#task: /;
    \*type  trace_elt.task_name;
    \*type  \# action: /;
    \*type  trace_elt.act;
    \*type  \# time: /;
    \*type  trace_elt.time;
    \*type  \# misc1 : /;
    \*type  trace_elt.misc1;
    \*type  \# misc2 : /;
    \*type  trace_elt.misc2;
    \*sizeq  curr_tcb.sts.resp_q, q.size;
  \*endwhile;
  \*type  \#/ elapsed time: /;
  \*type  \#/ resp_time;

472
PROCEDURE run_rtr_task
:
This procedure takes the first ready to run task off the rtr queue (this is the one with the highest priority) and copies its saved state into the current processor state. If there are no ready to run tasks, then an error message is printed and the kernel terminates.

PROC run_rtr_task;
*time ss.start;
pop taskstk, m_num;
sizeq rtr, br_chk;

if (br_chk = #0);
    printer */ERROR: NO RTR PROCESSES/;
    halt;
else;

    time cur_time;
    sub curr_tcb.date, cur_time, slack1;
    sub slack1, curr_tcb.resp_length, slack1;
    add curr_tcb.elap, slack1, slack1;
    if (curr_tcb.name = #/idle/);
        move #$0000, slack1;
    endif;
    if (m_num = #0);
        move #$00000, slack1;  DON'T WANT CURR_TASK TO RUN
    endif;
    move #/nil/, best_tcb.name;
    move #0, g_found;
sizeq rtr, g_q_size;
while (g_q_size > #0);
    delq rtr, try_tcb;
    if (try_tcb.name = #/idle/);
        move #$0000, slack2;
    else;
        sub try_tcb.date, cur_time, slack2;
        sub slack2, try_tcb.resp_length, slack2;
        add try_tcb.elap, slack2, slack2;
    endif;
    if (slack2 < slack1);
        move #1, g_found;
        move slack2, slack1;
        if (best_tcb.name <> #/nil/);
            addq best_tcb, rtr, #1;
        endif;
        move try_tcb, best_tcb;
    else;
        addq try_tcb, rtr, #1;
    endif;
    g_q_size, #1, g_q_size;
endwhile;
if (g_found = #1);
    add #1, preempts, preempts;

if (m_num = #1);
    pop taskstk, ret_pc;
    pop taskstk, ret_pc
    pop taskstk, ret_pc;
    move ret_pc, curr_tcb.pmt;
    move taskstk, curr_tcb.stk;
*add time curr_tcb.sts.finish;
    +sub curr_tcb.sts.finish, curr_tcb.sts.begin, curr_tcb.sts.begin;
*add curr_tcb.sts.begin, curr_tcb.sts.run_time, curr_tcb.sts.run_time;

473
move curr_tcb sts run_time, G_run_str;
+time curr_tcb sts begin;

move #/ready, curr_tcb status;
addq curr_tcb, ptr, &1;
endif;

move best_tcb name, tcb_name;
move best_tcb, curr_tcb;

add &1, switches, switches;
+time curr_tcb sts finish;
+sub curr_tcb sts finish, curr_tcb sts begin, curr_tcb sts begin;
+add curr_tcb sts begin, curr_tcb sts rdy_time, curr_tcb sts rdy_time;

move #/rdy_tm, rdy_str;
+move curr_tcb sts rdy_time, &rdy_str;
move curr_tcb sts wt_time, &wt_str;
+time curr_tcb sts begin;

move #/running/, curr_tcb status;
move curr_tcb stk, taskstk;
+time end os;
+sub end os, start os, elapsed;
+add elapsed, os ovhd, os ovhd;
+time sc_end;
+sub sc_end, sc start, sc start;
+add sc start, sc ovhd, sc ovhd;
enable &10;
move curr_tcb pont, pc;
endif;
ENDIF;
ENDPROC;

; PROCEDURE sched_rtr_task (rtr_pcb : t pcb)
;
; This procedure accepts a process control block and adds it to
; the ready to run queue in a prioritized fashion.
PROC sched_rtr_task;
+time sc_start;
pop taskstk, rtr_pcb;
move #/ready/, rtr_pcb status;
+time rtr_pcb sts finish;
+sub rtr_pcb sts finish, rtr_pcb sts begin, rtr_pcb sts begin;
+add rtr_pcb sts begin, rtr_pcb sts wt_time, rtr_pcb sts wt_time;
+move taskstk, tmp stk;
+time rtr_pcb sts begin;
addq rtr_pcb, ptr, &1;
+time sc_end;
+sub sc_end, sc start, sc start;
+add sc start, sc ovhd, sc ovhd;

; PROCEDURE get_sh (taskname : string)
;
; This procedure accepts a task name and sets the shared and
; sh trc variables to hold the names of the task's messaging and trace
; structures.
PROC get_sh;
pop taskstk, tempstr;
move &0, g found;
lock;
siseq comm_q, g q size;
while (g q size > &0) & (g found);
delq comm_q, g st els;
if (g st els.com = tempstr);
move &1, g found;
endif;
addq g st els, comm_q, &1;
sub g q size, &1, g q size;
endwhile;
unlock;
if (g found = &1);
move g st els st name, shared;
move g st els trc name, sh trc;
else;
move nil string, shared;

474
move nil_string, sh_trc;
endif;
ENDPROC;

; PROCEDURE get_local (taskname : string)
; This procedure accepts a task name and sets the local and
; loc_trc variables to hold the names of the task's messaging and trace
; structures.
PROC get_local;
pop taskstk, tempstr;
move $0, g_found;
sizeq loc_comm_q, g_q_size;
while (g_q_size > $0) & ('g_found');
delq loc_comm_q, g_st_elts;
if (g_st_elts.name = tempstr);
move $1, g_found;
endif;
addq g_st_elts, loc_comm_q, $1;
sub g_q_size, $1, g_q_size;
endwhile;
if (g_found = $1);
move g_st_elts.st_name, local;
move g_st_elts.trc_name, loc_trc;
else;
move nil_string, local;
move nil_string, loc_trc;
endif;
ENDPROC;

; PRIMITIVE nilprim
; This dummy primitive does nothing except pull in internode
; messages.
PRIM nilprim;
disable $10;
*time start_es;
*move curr_tcb.ste.ready_time, @ready_str;
*move curr_tcb.ste.s_time, @set_str;
*move curr_tcb.ste.run_time, @run_str;
if (num_queued > $0);
move $0, done;
sizeq loc_comm_q, q_size;
while (q_size > $0) & ('done');
delq loc_comm_q, st_elts;
move st_elts.st_name, local;
call get_sh, st_elts.nom;
lock;
while ($gshared.num_msgs > $0);
sub num_queued, $1, num_queued;
sub $gshared.num_msgs, $1, $gshared.num_msgs;
delq $gshared.msg, msg_elts;
if ($glocal.waiting = $1);
if ($glocal.st_type = msg_elts.type);
addq msg_elts, $glocal.msg,q, $q;
while (msg_elts.trc_size > $0);
*delq $gshared.trcs, trace_elts;
*addq trace_elts, $glocal.trcsize, $2;
*sub msg_elts.trc_size, $1, msg_elts.trc_size;
*endwhile;
else;
add msg_elts, $glocal.msg,q, $1;
while (msg_elts.trc_size > 0);
*delq $gshared.trcs, trace_elts;
*addq trace_elts, $glocal.trcsize, $1;
*sub msg_elts.trc_size, $1, msg_elts.trc_size;
*endwhile;
endif;
if ($glocal.st_type = $0) & ($glocal.st_type = msg_elts.type);
move $0, $glocal.waiting;
move msg_elts.due, $glocal.block.ddate;
move msg_elts.expires, $glocal.block.exp;
move msg_elts.length, $glocal.block.resp_length;
move $1, done;
endif;
endif;
endwhile;
unlock;
addq $t_1, loc_comm_q, $1;
sub $q_ms, $1, $q_ms;
endwhile;
if ($done = $1);
call sched_str_task, $local.block;
endif;
endif;
enable $10;
ENDPRIM;

; PRIMITIVE send (@task_name : string; @msg : string)
; This primitive accepts the name of a destination task and a
; message pointer. It then performs a nonblocking send operation.
PRIM send;
	disable $10;
	add $1, sends, sends;
	*1 time start_os;
	pop taskstk, msg_str;
	pop taskstk, dest_str;

call get_local, dest_str;
if ($local <> nil_string);" if ($local.waiting = $1);
if ($local.wait_type = msg_str.type);
	call response_update, msg_str, loc_trc, $2, $1;
	addq $msg_str, $local.msg_q, $2;
else;
	call response_update, msg_str, loc_trc, $1, $1;
	addq $msg_str, $local.msg_q, $1;
endif;
if (($local.wait_type = $0) | ($local.wait_type = msg_str.type));
move $0, $local.waiting;
move $msg_str.due, $local.block.ddate;
move $msg_str.elapsed, $local.block.elap;
move $msg_str.length, $local.block.resp_length;
call sched_str_task, $local.block;
endif;
else;
	call response_update, msg_str, loc_trc, $1, $1;
	addq $msg_str, $local.msg_q, $1;
endif;
else;
call get_sh, dest_str;
lock;
	add $1, num_queue, num_queue;

call response_update, msg_str, sh_trc, $1, $1;

dqq $msg_str, $shared.msgs, $1;
	add $1, $shared.num_msgs, $shared.num_msgs;
unlock;
endif;
*1 time end_os;
sub end_os, start_os, elapsed;
add elapsed, os_cvhd, os_cvhd;
enable $10;
ENDPRIM;

; PRIMITIVE recv (msg_type : integer; @msg : string)
; This primitive performs a blocking selective receive operation.
; It accepts the type of message expected ($0 means accept any type)
; and a pointer to where the message is to be placed. If a
; specific receive type is specified, the task is blocked until
; that type is available. Otherwise, the calling task is blocked
; until any message becomes available.
PROC block_rcv;
time curr_tcb_sth.finish;
sub curr_tcb_sth.finish, curr_tcb_sth.begin, curr_tcb_sth.begin;
add curr_tcb_sth.begin, curr_tcb_sth.run_time, curr_tcb_sth.run_time;
move curr_tcb_sth.run_time, $run_str;
time curr_tcb_sth.begin;

476
move msg_type, @localontvangst;
move #1, @localwaiting;
move msg_str, curr_tcb.alt_str;
move #/err/, curr_tcb.status;
move taskstk, curr_tcb.stk;
add #2, po, curr_tcb.punt;
move curr_tcb, @local.block;
call run_tcb_task, #0;
disable #10;
move curr_tcb.alt_str, msg_str;
call get_local, currtask;
move $0, @local.waiting;
sizeq @local.msg.q, done;
if (done > $0);
delq @local.msg.q, msg_str;
call response_update, msg_str, loc_trc, #2;
endif;
EINPAGC;

PREM recv;
disable #10;
@time start_q;
pop taskstk, msg_str;
pop taskstk, msg_type;
call get_local, currtask;
call get_sh, currtask;
while ($shared.num_msgs > $0);
lock;
sub num_queueed, #1, num_queueed;
sub $shared.num_msgs, #1, $shared.num_msgs;
delq $shared.msgs, msgلت;
addq msg. lett, $local.msg.q, #1;
while (msg.lett.trc.size > $0);
addq $shared.trace, trace.lett;
addq trace.lett, $local.trace,
sub msg.lett.size, #1, msg.lett.size;
endwhile;
unlock;
endwhile;
move $0, found;
sizeq @local.msg.q, q.size;
if (q.size = $0);
call block_recv;
else;
move #1, found;
if (msg.type <> $0);
; reorder msg queue because receive specific
move nil_queue, temp.q;
move nil_queue, temp2.q;
move $0, found;
sizeq @local.msg.q, m_num;
while ($num > $0);
delq @local.msg.q, msg.lett;
if (found) & (msg.lett.type = msg.type);
move #1, found;
addq msg. lett, temp.q, #2;
while (msg.lett.trc.size > $0);
delq @local.trace, trace.lett;
addq trace.lett, temp2.q, #2;
sub msg.lett.size, #1, msg.lett.size;
endwhile;
else;
addq msg. lett, temp.q, #1;
while (msg.lett.trc.size > $0);
delq @local.trace, trace.lett;
addq trace.lett, temp2.q, #1;
sub msg.lett.size, #1, msg.lett.size;
endwhile;
endif;
sizeq @local.msg.q, m_num;
endif .
move temp.q, @local.msg.q;
move temp2.q, @local.trace;
endif;
if (found);
call block_recv;

477
; PRIMITIVE sleep
; This primitive simply blocks the calling task by putting it in
; the sleep wait queue.
PRIM sleep;
  disable $10;
  *time start_cs;
  pop taskstk, ret_pc;
  *time curr_tcb.sts.finish;
  *sub curr_tcb.sts.finish, curr_tcb.sts.begin, curr_tcb.sts.begin;
  *add curr_tcb.sts.begin, curr_tcb.sts.run_time, curr_tcb.sts.run_time;
  *move curr_tcb.sts.run_time, $run_str;
  *time curr_tcb.sts.begin;
  move 0, /wait, curr_tcb.status;
  move taskstk, curr_tcb.stk;
  move ret_pc, curr_tcb.pcnt;
  addq curr_tcb, wait, curr_tcb.priority;
  call run_rst_task, $0;
ENDPRIM;

; PRIMITIVE get_data (task_name : string; data_name : string; size : integer)
; This primitive simulates data access between tasks. If the
; specified source task is not resident on the processor, then a number
; of shared accesses are made. One access is made for every 100 bytes
; of data requested.
PRIM get_data;
  disable $10;
  *time start_cs;
  pop taskstk, data_size;
  pop taskstk, msg_str;
  pop taskstk, dest_str;
  call get_local, dest_str;
  if (local = nil_string);
    while (data_size > #0);
      lock;
      move msg_str, q_size;
      unlock;
      sub data_size, #100, data_size;
      endwhile;
  else;
    while (data_size > #0);
      move q_size, q_size;
      sub data_size, #100, data_size;
      endwhile;
  endif;
  *time end_cs;
  *sub end_cs, start_cs, elapsed;
  *add elapsed, os_ovhd, os_ovhd;
  enable $10;
ENDPRIM;

; PRIMITIVE endResp
; This primitive simply calls response update to print the
; response traces.
PRIM endResp;
  *disable $10;
  *pop taskstk, m_num;
  *pop taskstk, resp_time;
  *call response_update, resp_time, m_num, #5;
  enable $10;
ENDPRIM;

478
; PRIMITIVE neu_resp
;
; This primitive clears the calling task's current trace queue.
PRIM neuresp;
*disable $10;
*sizeq curr_tcb.sta.resp.q, q.size;
*while (q.size > 0);
*delq curr_tcb.sta.resp.q, trace_elt;
*sizeq curr_tcb.sta.resp.q, q.size;
*endwhile;
*enable $10;
ENDPRIM;

; PRIMITIVE evt_synch
;
; This primitive serves as the interrupt handler for the synchronization
; primitive. This interrupt is raised periodically every 5 seconds.
PRIM evt_synch;
*disable $10;
*ackint $0;
*add $1, evt_0, evt_0;
*time start_os;
*sizeq await, wt.chk;
move $0, found;
while (((wt.chk > 0) & (~found)));
*delq await, temp_tcb;
if (temp_tcb.name = $/synch/);
*move $1, found;
time ch.syn.due;
*add $900, ch.syn.due, ch.syn.due;
*move 279, ch.syn.len;
*move ch.syn.due, temp_tcb.ddate;
*move $0, temp_tcb.elsap;
*move ch.syn.len, temp_tcb.resp_len;
*add $4, os.ovhd, os.ovhd;
else;
*add temp_tcb, await, temp_tcb.priority;
*sub wt.chk, $1, wt.chk;
*endif;
*endwhile;
*move curr_tcb, temp_tcb;
*move temp_tcb, curr_tcb;
*call response_update, $2, $8;
*move curr_tcb, temp_tcb;
*move temp_tcb, curr_tcb;
*call sched_rte_task, temp_tcb;
*endif;
*time end_os;
*sub end.os, start.os, elapsed;
*add elapsed, os.ovhd, os.ovhd;
*enable $10;
ENDPRIM;

; PRIMITIVE evt_loss_synch
;
; This primitive serves as the interrupt handler for the synchronization
; loss primitive. This interrupt is raised sporadically to signal a loss of
; synchronization.
PRIM evt_loss_synch;
*disable $10;
*ackint $1;
*add $1, evt_1, evt_1;
*time start_os;
move $1, synch_loss;
*sizeq await, wt.chk;
move $0, found;
while (((wt.chk > 0) & (~found)));
*delq await, temp_tcb;
if (temp_tcb.name = $/synch/);
*move $1, found;
time l.syn.due;
*add $700, l.syn.due, l.syn.due;
*move $64, l.syn.len;
move l.syn.due, temp_tcb.ddate;
move %0, temp_tcb.elap;
move $1, syn_len, temp_tcb.resp_len;
add $6, sc_ovhd, sc_ovhd;  
else;
add $1, temp_tcb.wait, temp_tcb.priority;
sub $1, sync, $1, sync;
endif;
endwhile;
if (found);
move curr_tcb, temp_tcb;
move temp_tcb, curr_tcb;
*call response_update, $1, $8;
move curr_tcb, temp_tcb;
move temp_tcb, curr_tcb;
*call sched_rtvz_task, temp_tcb;
endif;
*time end_os;
*sub end_os, start_os, elapsed;
*add elapsed, os_ovhd, os_ovhd;
enable $10;
ENDPRIM;

PRIM evt_msg_rx;

disable $10;
ackint $2;
*add $1, env2, env2;
*time start_os;
move $1, msg_rdy;
rnd $10, wt_chk;
if (wt_chk < $20000000)
move $1, msg_type;
else;
move $2, msg_type;
else;
move $3, msg_type;
endif;
endif;
mov $0, done;
call get_local, #msg/;
if ($local.waiting = $1);
mov $0, $local.waiting;
*move curr_tcb, temp_tcb;
*move $local.block, curr_tcb;
*call response_update, $2, $8;
time msg_due;
if (msg_type = $1);
add $50000, msg_due, msg_due;
mov $99, msg_len;
mov msg_due, curr_tcb.ddate;
mov $0, curr_tcb.elp;
mov msg_len, curr_tcb.resp_len;
add $4, sc_ovhd, sc_ovhd;
else;
if (msg_type = $2);
add $30000, msg_due, msg_due;
mov $1031, msg_len;
mov msg_due, curr_tcb.ddate;
mov $0, curr_tcb.elp;
mov msg_len, curr_tcb.resp_len;
add $4, sc_ovhd, sc_ovhd;
else;
add $12000, msg_due, msg_due;
mov $1569, msg_len;
mov msg_due, curr_tcb.ddate;
mov $0, curr_tcb.elp;
mov msg_len, curr_tcb.resp_len;
add $4, sc_ovhd, sc_ovhd;
endif;
endif;
move curr_tcb, $local.block;
*move temp_tcb, curr_tcb;
move $1, done;
endif;
if (done = #1);
    call sched_rtr_task, @local.block;
endif;
*time end_cs;
*sub end_cs, start_cs, elapsed;
*add elapsed, cs_ovhd, cs_ovhd;
enable #10;
ENDPRIM;

; PRIMITIVE evt_signal
; This primitive serves as the interrupt handler for the signalling
; interrupt handler. This interrupt is raised sporadically to signal the
; arrival of a signalling event. It makes the SIGNAL task ready to run
; from the blocked receive state.
PRIM evt_signal;
    disable #10;
    ackint #3;
    *add #1, evt_3, evt_3;
    *time start_cs;
    move #1, sig.rdy;
    rand #10, wt_chk;
    if (wt_chk < #5);
        move #6, sigtype;
    else;
        if (wt_chk < #8);
            move #4, sigtype;
        else;
            move #5, sigtype;
        endif;
    endif;
    move #0, done;
    call get_local, #/signal/;
    if (#local.waiting = #1);
        move #0, #local.waiting;
    *move curr_tcb, temp2_tcb;
    *move @local.block, curr_tcb;
    *call response_update, #3, #6;
    *time sig_due;
    if (sigtype = #6);
        add #1000, sig.due, sig_due;
        move #1200, sig.len;
        move sig.due, curr_tcb.ddate;
        move #0, curr_tcb.elap;
        move sig.len, curr_tcb.resp_len;
        *add #4, sc_ovhd, sc_ovhd;
    else;
        if (sigtype = #4);
            add #7000, sig.due, sig_due;
            move #2100, sig.len;
            move sig.due, curr_tcb.ddate;
            move #0, curr_tcb.elap;
            move sig.len, curr_tcb.resp_len;
            *add #4, sc_ovhd, sc_ovhd;
        else;
            add #2000, sig.due, sig_due;
            move #2120, sig.len;
            move sig.due, curr_tcb.ddate;
            move #0, curr_tcb.elap;
            move sig.len, curr_tcb.resp_len;
            *add #4, sc_ovhd, sc_ovhd;
        endif;
    endif;
    *move curr_tcb, @local.block;
    *move temp2_tcb, curr_tcb;
    move #1, done;
endif;
if (done = #1);
    call sched_rtr_task, @local.block;
endif;
*time end_cs;
*sub end_cs, start_cs, elapsed;
*add elapsed, cs_ovhd, cs_ovhd;
enable #10;
ENDPRIM;
This primitive serves as the interrupt handler for the clock interrupt
used as a timing base for the audits task. This is a periodic interrupt.

PRIM evt_clk;

disable $10;
ackint #8;
*add $1, evt_4, evt_4;
time start_os;
sleep waits, wt_clk;
move #0, found;
while ((wt_clk > #0) & (~found));
delay wait, temp_tcb;
if (temp_tcb.name = #/audits/);
move $1, found;
else;
add temp_tcb, wait, temp_tcb.priority;
sub wt_clk, #1, wt_clk;
endif;
endwhile;

if (found);
time end_due;
add #0100, end_due, end_due;
move #220, tim_len;
move #066, diag_len;
move end_due, temp_tcb.ddate;
move #0, temp_tcb.ellap;
move tim_len, temp_tcb.resp_len;
add #6, sc_ovhd, sc_ovhd;
move curr_tcb, temp_tcb;
move temp_tcb, curr_tcb;
call response_update, #4, #6;
move curr_tcb, temp_tcb;
move temp_tcb, curr_tcb;
call sched_tr_task, temp_tcb;
endif;
time end_os;
sub end_os, start_os, elapsed;
add elapsed, os_ovhd, os_ovhd;

enable $10;
ENDPRIM;

BEGIN
move #/model/, node_name;
disable $10;
move #/l.ti/, st_elts.st_name;
move #/l.ti.traces/, st_elts.trc_name;
addt st_elts, loc_comm_q, $#1;
move #/l.t2/, st_elts.st_name;
move #/l.t2.traces/, st_elts.trc_name;
addt st_elts, loc_comm_q, $#1;
move #/l.b/, st_elts.st_name;
move #/l.b.traces/, st_elts.trc_name;
addt st_elts, loc_comm_q, $#1;
move #/l.b/, st_elts.st_name;
move #/l.b.traces/, st_elts.trc_name;
addt st_elts, loc_comm_q, $#1;
move #/l.b/, st_elts.st_name;
move #/l.b.traces/, st_elts.trc_name;
addt st_elts, loc_comm_q, $#1;
move #/l.b/, st_elts.st_name;
move #/l.b.traces/, st_elts.trc_name;
addt st_elts, loc_comm_q, $#1;
move #/l.b/, st_elts.st_name;
move #/l.b.traces/, st_elts.trc_name;
addt st_elts, loc_comm_q, $#1;
move #/l.b/, st_elts.st_name;
move #/l.b.traces/, st_elts.trc_name;
addt st_elts, loc_comm_q, $#1;
move #/l.b/, st_elts.st_name;
move #/l.b.traces/, st_elts.trc_name;
addt st_elts, loc_comm_q, $#1;
move #/l.b/., st_elts.st_name;
move #/l.b.traces/., st_elts.trc_name;
addt st_elts, loc_comm_q, $#1;
move #/l.b/., st_elts.st_name;
move #/l.b.traces/., st_elts.trc_name;
addt st_elts, loc_comm_q, $#1;
move #/l.b/., st_elts.st_name;
move #/l.b.traces/., st_elts.trc_name;
addt st_elts, loc_comm_q, $#1;

addq st_el, loc_comm_q, $1;
move #/l_t12/, st_el, st_name;
addq #/l_t12, trc/, st_el, trc_name;
addq st_el, loc_comm_q, $1;
move #/l_t13/, st_el, st_name;
addq #/l_t13, trc/, st_el, trc_name;
addq st_el, loc_comm_q, $1;
move #/l_t14/, st_el, st_name;
addq #/l_t14, trc/, st_el, trc_name;
addq st_el, loc_comm_q, $1;
move #/l_t15/, st_el, st_name;
addq #/l_t15, trc/, st_el, trc_name;
addq st_el, loc_comm_q, $1;
move #/l_t16/, st_el, st_name;
addq #/l_t16, trc/, st_el, trc_name;
addq st_el, loc_comm_q, $1;
move #/l_t17/, st_el, st_name;
addq #/l_t17, trc/, st_el, trc_name;
addq st_el, loc_comm_q, $1;

if (mod_name = #/node4/);
lock;
move #/sh_t1/, st_el, st_name;
move #/sh_t1, trc/, st_el, trc_name;
addq st_el, comm_q, $1;
move #/sh_t2/, st_el, st_name;
addq #/sh_t2, trc/, st_el, trc_name;
addq st_el, comm_q, $1;
move #/sh_t3/, st_el, st_name;
addq #/sh_t3, trc/, st_el, trc_name;
addq st_el, comm_q, $1;
move #/sh_t4/, st_el, st_name;
addq #/sh_t4, trc/, st_el, trc_name;
addq st_el, comm_q, $1;
move #/sh_t5/, st_el, st_name;
addq #/sh_t5, trc/, st_el, trc_name;
addq st_el, comm_q, $1;
move #/sh_t6/, st_el, st_name;
addq #/sh_t6, trc/, st_el, trc_name;
addq st_el, comm_q, $1;
move #/sh_t7/, st_el, st_name;
move #/sh_t7, trc/, st_el, trc_name;
addq st_el, comm_q, $1;
move #/sh_t8/, st_el, st_name;
move #/sh_t8, trc/, st_el, trc_name;
addq st_el, comm_q, $1;
move #/sh_t9/, st_el, st_name;
move #/sh_t9, trc/, st_el, trc_name;
addq st_el, comm_q, $1;
move #/sh_t10/, st_el, st_name;
move #/sh_t10, trc/, st_el, trc_name;
addq st_el, comm_q, $1;
move #/sh_t11/, st_el, st_name;
move #/sh_t11, trc/, st_el, trc_name;
addq st_el, comm_q, $1;
move #/sh_t12/, st_el, st_name;
move #/sh_t12, trc/, st_el, trc_name;
addq st_el, comm_q, $1;
move #/sh_t13/, st_el, st_name;
move #/sh_t13, trc/, st_el, trc_name;
addq st_el, comm_q, $1;
move #/sh_t14/, st_el, st_name;
move #/sh_t14, trc/, st_el, trc_name;
addq st_el, comm_q, $1;
move #/sh_t15/, st_el, st_name;
move #/sh_t15, trc/, st_el, trc_name;
addq st_el, comm_q, $1;
move #/sh_t16/, st_el, st_name;
move #/sh_t16, trc/, st_el, trc_name;
addq st_el, comm_q, $1;
move #/sh_t17/, st_el, st_name;
move #/sh_t17, trc/, st_el, trc_name;
addq st_el, comm_q, $1;

unlock;
endif;
move tasklist, proclist;
sizeq proclist, br_chk;
move currtask, tempstr;
move #1, wt_chk;

while br_chk <> #0;
deq proclist, tcb_name;
cxew tcb_name;
time curr_tcb.stat.begin;

483
+move ut_chk, curr_tcb.num;
+add $1, ut_chk, ut_chk;

if (tcb_name <> \#/idle\#);
lock:
+stype nod_names;
delq loc_comm_q, st_elt;
mov tcb_name, st_elt.nam;
+stype \#/--LOCAL-- TASK: \#;
+stype tcb_name;
+stype \#/ QUEUE: \#;
+styp scr st_elt.st_name;
addq st_elt, temp_q, $1;
unlock;

lock:
delq comm_q, st_elt;
while (st_elt.nam <> nil_string);
  addq st_elt, comm_q, $1;
delq comm_q, st_elt;
endwhile;
mov tcb_name, st_elt.nam;
+stype \#/GLOBAL-- TASK: \#;
+stype tcb_name;
+stype \#/ QUEUE: \#;
+styps cr st_elt.st_name;
addq st_elt, comm_q, $1;
unlock;
endif;
mov curprio, curr_tcb.priority;
mov $20000, curr_tcb clans;
mov $0, curr_tcb.ela;
mov $1, curr_tcb.resp_len;
mov taskstr, curr_tcb.pwpt;
mov tcb_name, curr_tcb.name;
mov $ready, curr_tcb.status;
mov taskstk, curr_tcb.stk;
addq curr_tcb, ttr, $1;
alsoq prerlist, br_chk;
endwhile;
mov temp_q, loc_comm_q;
mov nil queued, temp_q;
cxtrs tempstr;
mov $/rdy_tm, rdy_str;
mov $/vt_tm, vt_str;
mov $/run_tm, run_str;
+move $/END RESPONSE ................../, end_str;
bind evt, synch, $0;
bind evt, lose_synchron, $1;
bind evt, msg_xx, $3;
bind evt, signal, $3;
bind evt, clk, $6;

enable $10;
call run, ttr, task, $0;
ENDTASK;
ENDERROF;
B.4.10  Least Slack Time Application Tasks

The application tasks for the least slack time system are similar to those used in the earliest due date scheduling. However, there are a few changes in the way message tags are propagated. The MSG and DIAGS tasks used in the LST system are given here as an example for comparison with their counterparts in section B.4.8.

; MSG Task
;
; STAT rdy_tm, integer;
; STAT wt_tm, integer;
; STAT run_tm, integer;
VAR byte_cnt, integer; No. of message bytes transmitted or received
VAR msg_pending, integer;
VAR msg_out, t_msg_end;
VAR msg_in, t_msg_end;
VAR tsk_strt, integer;
VAR tsk_end, integer;
STAT cycle, integer;
STAT num_cycles, integer;
VAR tot_time, integer;

ACTIVITY init:
  time tsk_strt;
  move #0, msg_pending;
  move #0, msg_in.type;
ENDACT;

ACTIVITY upd_cycle:
  move #0, msg_in.type;
  *add #1, num_cycles, num_cycles;
  time tsk_strt;
ENDACT;

ACTIVITY init_cnt:
  move #0, byte_cnt;
  move #0, msg_end;
ENDACT;

ACTIVITY incr_cnt:
  add #1, byte_cnt, byte_cnt;
ENDACT;

ACTIVITY send_tone:
  move #msg, msg_out.src;
  move #2, msg_out.type;
  move msg_due, msg_out.due;
  move msg_len, msg_out.length;
  time tsk_end;
  sub tsk_end, tsk_strt, msg_out.elapsed;
  *add #5, sc_ovhd, sc_ovhd;
ENDACT;

ACTIVITY send_start:
  move #1, msg_out.type;
  move #msg, msg_out.src;
  move msg_due, msg_out.due;
  move msg_len, msg_out.length;
  time tsk_end;
  sub tsk_end, tsk_strt, msg_out.elapsed;
  *add #5, sc_ovhd, sc_ovhd;
ENDACT;

ACTIVITY send_diag:
  move #3, msg_out.type;
  move #msg, msg_out.src;
  move msg_due, msg_out.due;
  move msg_len, msg_out.length;
  time tsk_end;

sub tsk_end, tsk_strt, msg_out elapsed;
*add #5, sc_ovhd, sc_ovhd;
ENDACT;

ACTIVITY ack_in:
  if (msg_in.type <> #0);
  move #1, msg_pending;
  endif;
ENDACT;

ACTIVITY clr_msg:
  move #0, msg_in.type;
  move #0, msg_pending;
ENDACT;

ACTIVITY set_sync:
  ticks #200;
  move #0, msg_in.type;
  move #0, msg_pending;
ENDACT;

BEGIN
DO.ACT init;
CALLP nilprim;
SETSTATE #/clear/;
LOOP:
  STATESSELECT:
    STATE clear;
    IF #true;
      DO.ACT upd_cycle;
      CALLP nextrep;
      SETSTATE #/start/;
    ENDIF;
    ENDSTATE;

    STATE start;
    IF (msg_pending = #1);
      CALLP nilprim;
      SETSTATE #/handl_msg/;
    ENDIF;
    IF (msg_rdy = #1);
      DO.ACT init_cnt;
      CALLP nilprim;
      SETSTATE #/rxtx/;
    ENDIF;
    IF #true;
      CALLP recv, #0, #/msg_in/;
    ENDIF;
    ENDSTATE;

    STATE sigmsg;
    IF #true;
      DO.ACT sigmsg;
      CALLP nilprim;
      SETSTATE #/start/;
    ENDIF;
    ENDSTATE;

    STATE rxtx;
    IF (byte_cnt < #22);
      DO.ACT incr_cnt;
      CALLP nilprim;
      SETSTATE #/rxtx/;
    ENDIF;
    IF #true;
      CALLP nilprim;
      SETSTATE #/msg_fin/;
    ENDIF;
    ENDSTATE;

    STATE msg_fin;
    IF (msgtype = #1);
      DO.ACT send_start;
      CALLP send, #/cnf/; #/msg_out/;
      SETSTATE #/clear/;
    ENDIF;
    IF (msgtype = #2);
      DO.ACT send_tone;
      CALLP send, #/tone/; #/msg_out/;
SETSTATE #/clear/
ENDIF;
IF (msg.type = #3);
DO.ACT send_diag;
CALLP send, #/diag/, #/msg.out/;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;

STATE hand_mag;
IF (msg.in.type = #7);
DO.ACT clr_mag;
CALLP endresp, msg.in.elapse, #1;
SETSTATE #/clear/;
ENDIF;
IF (msg.in.type = #8);
DO.ACT set_synch;
CALLP endresp, msg.in.elapse, #2;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;
ENDSELECT;
ENDLOOP
ENDSTATE;

; DIAG Task
;
STATE rdty_tm, integer;
STATE wt_tm, integer;
STATE run_tm, integer;
VAR msg.out, t.msg.alt;
VAR msg_in, t.msg.alt;
VAR dest, string;
VAR dg_num, integer;
VAR tak_strt, integer;
VAR tak_end, integer;
VAR cycle, integer;
STATE num_cycles, integer;
VAR tot_time, integer;

ACTIVITY init;
+time tak_strt;
ENDACT;

ACTIVITY upd_cycle;
+add #1, num_cycles, num_cycles;
time tak_strt;
ENDACT;

ACTIVITY time_wt;
time tak_strt;
ENDACT;

ACTIVITY chnlreq;
time tak_strt;
move #12, msg_out.type;
move #/diag/, msg_out.src;
move msg.in.due, msg.out.due;
move msg.in.length, msg.out.length;
time tak_end;
sub tak_end, tak_strt, tak_end;
add tak_end, msg.in.elapse, msg_out.elapse;
end #0, sc.ovhd, sc_ovhd;
ENDACT;

ACTIVITY choose_diag;
time tak_strt;
rand #9, dg_num;
move #21, msg_out.type;
move #/diag/, msg_out.src;
if (dg_num = #2);
move #/mmdiag/, dest;
else;
move #/sidiag/, dest;
endif;
ENDIF;
end;
move msg.in.due, msg.out.due;
move msg.in.length, msg.out.length;
time tak_end;
sub tak_end, tak_strt, tak_end;
add tak_end, msg.in.elapse, msg_out.elapse;
end #0, sc_ovhd, sc_ovhd;
ENDACT;

BEGIN
DO.ACT init;
CALLP nilprim;
SETSTATE #/clear/;
LOOP;
STATESTATE;
STATE clear;
+upd_cycle;
DO.ACT
CALLP nextresp;
SETSTATE #/start/;
ENDIF;
ENDSTATE;
STATE start;
+if true;
CALLP recy, #2, #/msg.in/;
SETSTATE #/got_msg/;
ENDIF;
ENDSTATE;
STATE got_msg;
+if true;
DO.ACT chnl req;
CALLP send, #/chnl/, #/msg.out/;
SETSTATE #/wt_ack1/;
ENDIF;
ENDSTATE;
STATE wt_ack1;
+if true;
CALLP recy, #14, #/msg.in/;
SETSTATE #/got_chnl/;
ENDIF;
ENDSTATE;
STATE got_chnl;
+if true;
DO.ACT chnl req;
CALLP send, #/chnl/, #/msg.out/;
SETSTATE #/wt_ack2/;
ENDIF;
ENDSTATE;
STATE wt_ack2;
+if true;
CALLP recy, #14, #/msg.in/;
SETSTATE #/pick_diag/;
ENDIF;
ENDSTATE;
STATE pick_diag;
+if true;
DO.ACT choose_diag;
CALLP send, dest, #/msg.out/;
SETSTATE #/clear/;
ENDIF;
ENDSTATE;
ENDSELECT;
ENDLOOP
ENDTASK;