NOTICE

The quality of this microfiche 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 a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED

AVIS

La qualité de cette microfiche 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 mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

LA THÈSE A ÉTÉ MICROFILMÉE TELLE QUE NOUS L'AVONS RECUE
NAME OF AUTHOR/NOM DE L'AUTEUR: STEPHEN G. MICHEL

TITLE OF THESIS/TITRE DE LA THÈSE: HIERARCHICAL DESIGNS FOR LAYERED PROTOCOL SYSTEMS

UNIVERSITY/UNIVERSITÉ: CARLETON UNIVERSITY

DEGREE FOR WHICH THESIS WAS PRESENTED/GRÂDE POUR LEQUEL CETTE THÈSE FUT PRÉSENTÉE: MASTER OF SCIENCE (INFORMATION SYSTEMS)

YEAR THIS DEGREE CONFERRED/ANNÉE D'OBTENTION DE CE DÉGÎRE: 1982

NAME OF SUPERVISOR/NOM DU DIRECTEUR DE THÈSE: DR. R.J.A. BUHR

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

DATE/DATÉ: 21 SEP 87

SIGNED/SIGNÉ: [Signature]

PERMANENT ADDRESS/RÉSIDENCE FIXE: 124-1449 RIDGE BROOK DR.

[Address]

[Signature]
HIERARCHICAL DESIGNS FOR LAYERED PROTOCOL SYSTEMS

by

© STEPHEN G MICHELL

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfilment of the requirement
for the degree of
Master of Science

DEPARTMENT OF MATHEMATICS
FACULTY OF SCIENCE
CARLETON UNIVERSITY
OTTAWA, ONTARIO
The undersigned recommend to the
Faculty of Graduate Studies and Research
acceptance of the thesis

HIERARCHICAL DESIGNS FOR LAYERED PROTOCOL SYSTEMS

submitted by Stephen G. Michell
in partial fulfilment of the degree of
Master of Science

R. J. A. Buhr
Thesis Supervisor

K. S. Williams
Chairman
Department of Mathematics
and Statistics

Carleton University
September, 1982
ABSTRACT

This thesis investigates the use of structured design methodologies to specify and implement entities which implement layered protocols, such as those specified by the Open System Interconnection. The design approach begins with the analysis of data flow and the assignment of functional units to manage this data. From this data flow graph, structure graphs are created which assign the functional units to structure objects, and the data flow to interfaces between those units. Once the structured objects have been assigned, they are coded into Ada specifications, first the interface, then the code for the objects. From this design, implementations of layered protocol from the designs are examined, with a particular view to implementing such entities on existing systems.
ACKNOWLEDGEMENTS

Many of the ideas and developments contained in this thesis had their genesis in discussions with Dennis Mckinnon. His contribution is deeply appreciated. Special mention goes to Ray Buhr, whose constant encouragement and assistance helped steer me and helped me formulate and solidify all-too-frequent fuzzy concepts. His guiding hand was crucial to this effort.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>SECTION</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td></td>
<td>Acknowledgements</td>
<td>iv</td>
</tr>
<tr>
<td></td>
<td>Table of Contents</td>
<td>v</td>
</tr>
<tr>
<td></td>
<td>List of Figures</td>
<td>vi</td>
</tr>
<tr>
<td>1.1</td>
<td>Chapter 1 Introduction</td>
<td></td>
</tr>
<tr>
<td>1.1.1</td>
<td>Background</td>
<td>1-1</td>
</tr>
<tr>
<td>1.1.2</td>
<td>Design</td>
<td>1-1</td>
</tr>
<tr>
<td>1.2</td>
<td>Layered Protocol Systems</td>
<td>1-2</td>
</tr>
<tr>
<td>1.3</td>
<td>Objective</td>
<td>1-3</td>
</tr>
<tr>
<td>1.4</td>
<td>Activities And Accomplishments</td>
<td>1-3</td>
</tr>
<tr>
<td></td>
<td>Outline</td>
<td>1-4</td>
</tr>
<tr>
<td>2.1</td>
<td>Chapter 2 Logical Design Of Layered Protocol Systems</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>Introduction</td>
<td>2-1</td>
</tr>
<tr>
<td>2.2.1</td>
<td>The X.25 Layered Protocol Specification</td>
<td>2-2</td>
</tr>
<tr>
<td>2.3</td>
<td>Summary</td>
<td>2-5</td>
</tr>
<tr>
<td>2.3.1</td>
<td>X.25: A Study In System Design</td>
<td>2-8</td>
</tr>
<tr>
<td>2.3.1.1</td>
<td>Introduction</td>
<td>2-8</td>
</tr>
<tr>
<td>2.3.1.2</td>
<td>The Design Process</td>
<td>2-8</td>
</tr>
<tr>
<td>2.3.2</td>
<td>X.25 Protocol Data Units</td>
<td>2-10</td>
</tr>
<tr>
<td>2.3.3</td>
<td>External Structure Of An X.25 Service</td>
<td>2-10</td>
</tr>
<tr>
<td>2.3.3.1</td>
<td>X.25 Packet Layer Manager - Data Flow</td>
<td>2-10</td>
</tr>
<tr>
<td></td>
<td>Virtual Circuit Manager</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Internal Data Flow</td>
<td>2-13</td>
</tr>
<tr>
<td>2.3.3.2</td>
<td>Dispatcher - Internal Data Flow</td>
<td>2-15</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Allocation Of Structured Objects</td>
<td>2-16</td>
</tr>
<tr>
<td>2.3.4.1</td>
<td>Network Manager - External View</td>
<td>2-16</td>
</tr>
<tr>
<td>2.3.4.2</td>
<td>System Layering</td>
<td>2-17</td>
</tr>
<tr>
<td>2.3.5</td>
<td>Example Of A Network Layer Internal Organization</td>
<td></td>
</tr>
<tr>
<td>2.3.5.1</td>
<td>Assignment Of Objects</td>
<td>2-19</td>
</tr>
<tr>
<td>2.3.5.2</td>
<td>The Dispatcher</td>
<td>2-23</td>
</tr>
<tr>
<td>2.3.5.3</td>
<td>The Virtual Circuit Manager</td>
<td>2-25</td>
</tr>
<tr>
<td>2.3.5.4</td>
<td>The Router Function</td>
<td>2-34</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>SECTION</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.6</td>
<td>Summary And Conclusions</td>
<td>2-34</td>
</tr>
<tr>
<td>2.4</td>
<td>Another Look At Internal Layer Architecture</td>
<td></td>
</tr>
<tr>
<td>2.4.1</td>
<td>Alternate Designs And Issues</td>
<td></td>
</tr>
<tr>
<td>2.4.2</td>
<td>Monolithic Designs</td>
<td>2-35</td>
</tr>
<tr>
<td>2.4.2.1</td>
<td>The Single Manager Approach</td>
<td>2-36</td>
</tr>
<tr>
<td>2.4.2.2</td>
<td>The Design</td>
<td>2-36</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Issues In Design</td>
<td>2-39</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Functional Manager Design</td>
<td>2-43</td>
</tr>
<tr>
<td>2.4.4.1</td>
<td>The Design</td>
<td>2-43</td>
</tr>
<tr>
<td>2.4.4.2</td>
<td>Issues</td>
<td>2-45</td>
</tr>
<tr>
<td>2.4.5</td>
<td>Dynamic Manager Design</td>
<td>2-46</td>
</tr>
<tr>
<td>2.4.5</td>
<td>Conclusions</td>
<td>2-47</td>
</tr>
<tr>
<td>2.5</td>
<td>Extending The Layer Concept</td>
<td>2-49</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Frame-work Of Open System Interconnection</td>
<td>2-50</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Some Principles For Osi System Design</td>
<td>2-51</td>
</tr>
</tbody>
</table>

## Chapter 3

From The Iso Fdt To System Design

<p>| 3.1       | Introduction                                                          | 3-1  |
| 3.1.1     | Overview Of The Iso Proposed                                          |      |
|           | Formal Description Technique                                           | 3-1  |
| 3.1.2     | Place Of Formal Description in System Design                          | 3-3  |
| 3.2       | Using The Iso Formal Description Technique:                           |      |
|           | Bochmann’s Class O Transport Manager                                  | 3-4  |
| 3.2.1     | The Osi Formal Description Technique                                  | 3-4  |
| 3.2.2     | Discussion                                                            | 3-8  |
| 3.3       | Using The Iso Fdt: A Second Example                                   | 3-9  |
| 3.3.1     | Data Flow Graph Inspired Development Of A Skeleton Fdt                | 3-10 |
| 3.3.2     | Ada Design Of Multi-connection                                         |      |</p>
<table>
<thead>
<tr>
<th>TABLE OF CONTENTS</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECTION</td>
<td>TITLE</td>
<td>PAGE</td>
</tr>
<tr>
<td>Transport Entity</td>
<td></td>
<td>3-17</td>
</tr>
<tr>
<td>3.3.2.1</td>
<td>Data Flow</td>
<td>3-18</td>
</tr>
<tr>
<td>3.3.2.2</td>
<td>Structured Diagrams</td>
<td>3-18</td>
</tr>
<tr>
<td>3.3.2.3</td>
<td>Ada Specification</td>
<td>3-23</td>
</tr>
<tr>
<td>3.4</td>
<td>Mapping Fdt Constructs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>To Ada Design Constructs</td>
<td>3-27</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Fdt Modules</td>
<td>3-27</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Fdt Interactions</td>
<td>3-28</td>
</tr>
<tr>
<td>3.4.3</td>
<td>State Transitions</td>
<td>3-31</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Summary</td>
<td>3-32</td>
</tr>
<tr>
<td>3.5</td>
<td>Using The Iso Fdt: A Third Example</td>
<td>3-32</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Introduction</td>
<td>3-32</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Overview Of The Monolithic Entity</td>
<td>3-33</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Issues In Mapping Monolithic Specification</td>
<td>3-35</td>
</tr>
<tr>
<td></td>
<td>To Design</td>
<td></td>
</tr>
<tr>
<td>3.5.3.1</td>
<td>Isomorphic Designs</td>
<td>3-35</td>
</tr>
<tr>
<td>3.5.3.2</td>
<td>Attempt At A Non-isomorphic Design</td>
<td>3-39</td>
</tr>
<tr>
<td>3.6</td>
<td>Conclusions</td>
<td>3-41</td>
</tr>
</tbody>
</table>

Chapter 4 Further Issues In Osi Design

<p>| 4.1              | Introduction                               | 4-1  |
| 4.2              | Non-hierarchical Control Designs           | 4-1  |
| 4.2.1            | Osi Fdt Isomorphic Design                 | 4-1  |
| 4.2.1.1          | Mapping Interactions To Tasks             | 4-1  |
| 4.2.1.2          | A Message Oriented Design                 | 4-2  |
| 4.2.1.3          | A Piatkowski-derived Design               | 4-6  |
| 4.3              | Connection Management Issues              | 4-8  |
| 4.3.1            | General                                    | 4-8  |
| 4.3.2            | Dispatcher - Virtual Circuit Manager Design | 4-8  |
| 4.3.3            | Manager - Port Design                      | 4-9  |
| 4.3.4            | Discussion                                 | 4-10 |
| 4.4              | The Use Of Channels For                    | 4-10 |
|                  | Inter-module Communication                |      |</p>
<table>
<thead>
<tr>
<th>TABLE OF CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECTION</td>
</tr>
<tr>
<td>4.5</td>
</tr>
<tr>
<td>Chapter 5 Implementing An Osi Design</td>
</tr>
<tr>
<td>5.1</td>
</tr>
<tr>
<td>5.2</td>
</tr>
<tr>
<td>5.2.1</td>
</tr>
<tr>
<td>5.2.2</td>
</tr>
<tr>
<td>5.2.3</td>
</tr>
<tr>
<td>5.3</td>
</tr>
<tr>
<td>5.3.1</td>
</tr>
<tr>
<td>5.3.1.1</td>
</tr>
<tr>
<td>5.3.1.2</td>
</tr>
<tr>
<td>5.3.2</td>
</tr>
<tr>
<td>5.3.2.1</td>
</tr>
<tr>
<td>5.3.2.2</td>
</tr>
<tr>
<td>5.4</td>
</tr>
<tr>
<td>5.4.1</td>
</tr>
<tr>
<td>5.4.1.1</td>
</tr>
<tr>
<td>5.4.1.1</td>
</tr>
<tr>
<td>5.4.1.2</td>
</tr>
<tr>
<td>5.4.1.3</td>
</tr>
<tr>
<td>5.4.1.3</td>
</tr>
<tr>
<td>5.4.2</td>
</tr>
<tr>
<td>5.4.3</td>
</tr>
<tr>
<td>5.5</td>
</tr>
<tr>
<td>Chapter 6 Conclusions</td>
</tr>
<tr>
<td>6.1</td>
</tr>
<tr>
<td>6.2</td>
</tr>
<tr>
<td>6.3</td>
</tr>
<tr>
<td>NUMBER</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>2.1</td>
</tr>
<tr>
<td>2.2</td>
</tr>
<tr>
<td>2.3</td>
</tr>
<tr>
<td>2.4</td>
</tr>
<tr>
<td>2.5</td>
</tr>
<tr>
<td>2.6</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>2.7</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>2.8</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>2.9</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>2.10</td>
</tr>
<tr>
<td>2.11</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>2.12</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>2.13</td>
</tr>
<tr>
<td>2.14</td>
</tr>
<tr>
<td>2.15</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>2.16</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>2.17</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>2.18</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
</tr>
<tr>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>3.13 Ada Specification of Transport Entity</td>
</tr>
<tr>
<td>3.14 Ada Specification of Transport Entity</td>
</tr>
<tr>
<td>3.15 Implementing FDT Channel</td>
</tr>
<tr>
<td>3.16 Boehmann's Multiple Connection</td>
</tr>
<tr>
<td>3.17 Skeleton of Multiple Connection</td>
</tr>
<tr>
<td>3.18 Isomorphic Monolithic Manager</td>
</tr>
<tr>
<td>3.19 Isomorphic Monolithic Manager</td>
</tr>
<tr>
<td>3.20 Non-isomorphic Design from</td>
</tr>
<tr>
<td>4.1 Alternate Mappings of FDT Channels</td>
</tr>
<tr>
<td>4.2 Implementing a Sap Task</td>
</tr>
<tr>
<td>4.3 Interface Specification of a Sap Task</td>
</tr>
<tr>
<td>4.4 Example of a Sap Provider</td>
</tr>
<tr>
<td>4.5 A Piatkowski Inspired OSI Layer Design</td>
</tr>
<tr>
<td>5.1 Highly Distributed OSI Node</td>
</tr>
<tr>
<td>5.2 OSI Node Resident on Existing System</td>
</tr>
<tr>
<td>5.3 iRMX 80 Implementation of a Rendezvous</td>
</tr>
<tr>
<td>5.4 iRMX 80 Exchange Implementation of Rendezvous</td>
</tr>
<tr>
<td>5.5 PL/M Code Fragments of iRMX 80 Implementation</td>
</tr>
<tr>
<td>5.6 iRMX 80 Implementation</td>
</tr>
<tr>
<td>5.7 Mapping Ada Entries to</td>
</tr>
<tr>
<td>FIGURE</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>5.8</td>
</tr>
<tr>
<td>5.9</td>
</tr>
<tr>
<td>5.10</td>
</tr>
<tr>
<td>5.11</td>
</tr>
<tr>
<td>5.12</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>5.13</td>
</tr>
<tr>
<td>5.14</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

1.1 BACKGROUND

1.1.1 DESIGN

As systems have grown, become more complex and distributed, a clear requirement for good system design methods and languages has become evident. Yourdon and Constantine[16] have injected into this process graphical methods which assist the designer in conceptualizing the system. Up to the present, however, there has not been a method which addresses concurrent systems, nor which will treat hardware and software in a similar fashion, while structured diagram methods such as Yourdon's method lack a suitable design specification language to complement the intuitive designs expressed by the structured graphs.

There is a clear requirement for a design methodology, and languages which allow designs - distributed or monolithic, concurrent or sequential - to be clearly expressed. Such a method must also handle hardware or software in a similar manner, and in doing so, give the designer the ability to defer commitment of hardware, systems, etc. to the implementation stage.

A great deal of work has been done at Carleton recently on these issues. Buhr and the author[5] have developed graphical structure notation. This notation and design strategies associated with the use of structured diagrams and Ada as a specification language are contained in a draft
manuscript by Buhr[18]. Buhr and McKinnon[19] have used
this method to study issues in specifying a Transport Layer
Entity in the OSI. Boucouris[20] has used Ada to specify
the design of Data Flow architectures. A previous paper by
the author and Buhr uses the Ada programming language in
conjunction with this structure graph notation effectively
to specify the design of systems. The Ada language, with
its modular units (packages and tasks) and uniform access to
these units (procedures and entries) gives the freedom and
power of expression to permit it to be used effectively[14].

1.1.2 LAYERED PROTOCOL SYSTEMS

A great deal of effort is being spent by the Interna-
tional Organization for Standards to develop a reference
model for a layered protocol system called Open System In-
terconnect. Each layer within the model has complex proto-
cols associated with the communication and transactions
between peer "entities" in different nodes. It also pro-
vides for an interface between layers in one node via Ser-
vice Access Points, which provide an address for a user, but
also handle all service requests and results.

A great deal of work is being done to formally define
this reference model, and propose formal description techni-
cues for it. Piatkowski[13] and Bochmann[3,10] have both
proposed methods of formally specifying entities in a node
in the Open System Interconnection reference model. Both of
these methods are based on Pascal or extensions to Pascal.
The reference model proposed by Bochmann is intended to for-
mally describe the interfacing to, and functionality of en-
tities in the formal reference model. It is not intended to
direct or specify any particular design.

1.2 OBJECTIVE
The objective of this thesis was to:

1. examine the design of layered protocol systems, from formal description through to implementation, using hierarchical designs as a basis;

2. propose designs derived from the proposed OSI FDT specifications; and

3. investigate several implementations of this hierarchical, distributed design using systems available presently at Carleton for stand alone or co-resident systems.

1.3 ACTIVITIES AND ACCOMPLISHMENTS

The first activity was to investigate the use of Ada as a specification language for existing layered protocol. X.25 was chosen because it is an international standard, is layered and has already been implemented, both here at Carleton and commercially. This portion of the work resulted in a paper[5] which is to be presented at the Distributed Computing Conference in October, 1982.

The next activity was to investigate the ISO protocol specification formal description technique (FDT) based on extensions to the Pascal language. The approach was to attempt to design Transport Layer Entities based on sample specifications of them given by the OSI FDT[3,10]. In doing this, some weaknesses of the FDT were uncovered and its surprising closeness to system design was revealed. A paper on this by Buhr and the author is in preparation.

In creating design specifications using Ada, specifications of non-hierarchical designs were not examined in detail. This was mainly because Ada's inter-task communications
and procedural interfaces to packages are by nature hierarchical, and it was believed that first attempts at demonstrating the power of Ada as a design specification tool should stay close to natural Ada use and constructs.

The final activity examined potential implementations of the designs created in this thesis. No implementations were actually created, as this would have restricted the author to only one system, whereas the intent was to show the universality of the design approach: This required showing how to map these hierarchical designs on to both distributed and monolithic systems.

4.4 OUTLINE

This thesis proceeds by first showing how the proposed design techniques can be used by developing a hierarchical design of an X.25 packet switched service in chapter 2.

Chapter 3 then summarizes the impending OSI FDT technique and shows where it fits into the design process. It proceeds by developing design skeletons of alternate OSI Transport layer services based on these FDT specifications, and examines design issues that arise from these attempts.

Chapter 4 is a discussion chapter which examines in more depth some issues in the design of OSI nodes, and the design process in general.

Chapter 5 looks at several implementations of OSI designs on various systems for stand-alone use, and for co-resident use. Chapter 6 summarizes the thesis.

The final chapter shows how the objectives were met and evaluates the design. It also suggests for future work.
CHAPTER 2
LOGICAL DESIGN OF LAYERED PROTOCOL SYSTEMS

2.1 INTRODUCTION

This chapter shows how to logically design a communication service to interface with an X.25 packet-switched network according to the CCITT specification of the X.25 protocol[9]. The design method uses structural design techniques with the Ada programming language being the design specification language. The intent is not to study X.25 per se, but to use it as a concrete example of a layered protocol.

Bowen and Buhr[4] showed how to design an HDLC link protocol manager using monitors. Cavers[7], Bochmann[2] have presented alternate design methods for X.25 systems. Kaye[12] presented a distributed implementation of an X.25 packet switched service where the network layer resided on a PDP-11 processor running Multi-Pascal using the HDLC link services of an 8080 based system. This chapter revisits X.25, showing how to use high level design techniques as suggested by Buhr and McKinnon[19] and formally specifying the final design in the new D.OiD. programming language Ada. Several alternate designs and associated design issues are examined.

Section 2.2 discusses the nature of the X.25 communications protocol specification.

Section 2.3 develops a hierarchical control sub-system
design for an X.25 service.

Section 2.4 examines alternate designs of an X.25 node and uses them to consider more advanced issues and to discuss some of the issues and trade-offs involved in designing highly concurrent systems.

Section 2.5 discusses more elaborate layered protocols and proposes general design techniques for such systems, in particular, the design of nodes for Open System Interconnection.

2.2 THE X.25 LAYERED PROTOCOL SPECIFICATION

The X.25 packet switched network specification of the CCITT (International Telephone and Telegraph Communications Committee) specifies the communications protocol that a user's system must adhere to in order to use a commercial X.25 packet switched network to establish communication with remote systems. By following the X.25 protocol, a user system can establish many independent communication links with any number of "PROCESSES" in one or more other user systems connected to the network, as shown in Figure 2.1. This communication is in the form of "CALLS", or "VIRTUAL CIRCUITS"; all calls share the same physical link to the network, the same local "X.25 PROTOCOL MANAGER", and the same network resources.

The term "PROCESS" is used here to denote an executing user program which is functionally independent from the X.25 service. The term "TASK" are reserved for those concurrent activities which function co-operatively and intimately to achieve the function of the Entity.
FIGURE 2.1 OVERVIEW OF SYSTEM COMMUNICATING WITH X.25 NETWORK

The X.25 communications protocol establishes virtual circuits and manages them from call placement through acceptance, data transfer to clearing, and manages the transfer of data so that messages (packets) are delivered to the correct destination free of errors. To perform these functions, X.25 specifies two independent protocol management layers:

1. The frame level logical interface to manage the single link to the network and to ensure that frames, as they are called, are transmitted free of errors across the link, that frames received are acknowledged, and that frames sent but not acknowledged are resent; and,
2. The PACKET LEVEL LOGICAL INTERFACE to communicate with the network to ensure that communications are valid and to establish and manage each VIRTUAL CIRCUIT as described above.

According to the X.25 protocol, the PACKETS are carried inside FRAMES, as shown in figure 2.2. User's data is placed into packets of up to (typically) 128 or 256 bytes, and information as to the Virtual Circuit Number, type of packet, packet number (on that Virtual Circuit) etc. is added as a header to the data to form the PACKET. Link Control information, such as frame count, is then added as FRAME HEADER to the packet, and the frame is sent to the network over the link. Figure 2.3 shows in more detail the layout of a packet, while figure 2.4 gives an Ada specification equivalent to the X.25 specification of a packet.
2.2.1 Summary

Within each frame transferred across the link, there are actually (at least) three levels of communications protocol:

1. The user's data, which will contain some kinds of protocol, at least implicit, to manage its interactions with the end user;

2. The Network management protocol as described above in the Packet header; and,

3. The Link Control Protocol in the frame header.
FIG 2.3: PICTORIAL VIEW OF AN X.25 PACKET

X.25 addresses the specification of the last two protocol layers. It dictates:

1. The notion of a Virtual Connection;

2. The states that these Virtual Circuits can be in - cleared, call setup, data transfer, clearing;

3. The format of contents of the PACKETS exchanged;

4. The changes in state upon receipt of a packet, time-out, etc.;

5. The packets issued as a result of the aforementioned state changes;

6. The placement of these packets into FRAMES; and,
7. The (HDLC) link control protocol to manage the transfer of the frames.

X.25 does not specify how a user interacts with a local X.25 service, nor does it specify the actual design of a local service to implement its specification. The next section will address these issues when it examines the design of an X.25 service.

type CONTROL_MESSAGE_KIND is
  ('RECEIVE READY, INTERRUPT REQUEST, RECEIVE NOT READY,
   INTERRUPT CONFIRM, REJECT, CALL REQUEST, CONNECT CONFIRM,
   CLEAR REQUEST, CLEAR CONFIRM, RESTART OR RESET REQ,
   RESTART OR RESET CONFIRM);

for CONTROL_MESSAGE_KIND use
  RECEIVE READY => 0,
  INTERRUPT REQUEST => 1,
  RECEIVE NOT READY => 2,
  INTERRUPT CONFIRM => 3,
  REJECT => 4,
  CALL REQUEST => 5,
  CONNECT CONFIRM => 6,
  CLEAR REQUEST => 7,
  CLEAR CONFIRM => 8,
  RESTART OR RESET REQ => 9,
  RESTART OR RESET CONFIRM => 10;

end type;

type X_25_PACKET(kind: ?data, control)) is record
  HEADER: 0..16 CONST := 1;
  LOGICAL_CHANNEL_ID : 0 .. 7777B;
  case KIND is
    when DATA => RECEIVE COUNT, SEND_COUNT : integer;
    when CONTROL => RECEIVED_PACKET : 0 .. 7 := 0;
    when CONTROL_KIND => CONTROL_MESSAGE_KIND;
      case CONTROL_KIND is
        when CLEAR REQUEST => CAUSE : CL_CAUSE;
        when RESTART REQ => CAUSE : RESTART_CAUSE;
        when CLEAR REQ => CAUSE : CLEAR_CAUSE;
        when RESET REQ => CAUSE : FACILITIES;"
2.3 X.25: A STUDY IN SYSTEM DESIGN

2.3.1 INTRODUCTION

Now that X.25 has been introduced, and the packets exchanged have been specified, we can begin to logically design a local X.25 protocol manager to meet these specifications.

2.3.1.1 The Design Process –

The design process proceeds as follows:

1. Determine the major functional units of the design and draw them on a page as "Bubbles" or uncommitted modules;

2. Determine the major interactions between these "Bubbles" – shown in figure 2.5 as circles with arrows to indicate the direction of passage between the functional units;

3. Refine recursively the design of each functional unit by determining the functional units it needs to fulfill its function, and the data units exchanged internally;

4. Allocate system objects (packages, procedures and tasks) to the major functional units and define the interface specifications;

5. Repeat Step 4 recursively for all functional units created in step 3;

6. Code the objects for each functional unit.
FIG 2.5: DATA FLOW GRAPH OF X.25
- PACKET & LINK LEVELS COMMUNICATING OVER LINK
2.3.1.2 X.25 PROTOCOL DATA UNITS -

All of the communication protocols discussed in the previous section could be put into each user program wishing service from an X.25 system, but by designing an entity to manage the packet-level transactions, the idiosyncracies of the packet-level communication are hidden from the user. Instead, an application interacts with the network level manager in its node which performs the actual details of enveloping the message, transmission, reception, and stripping it, etc. for the user.

2.3.2 EXTERNAL STRUCTURE OF AN X.25 SERVICE

As was mentioned in the summary of the last section, there are three natural layers of protocols in each frame exchanged, the USERS LAYER, the NETWORK LAYER, and the LINK LAYER. These three layers of protocol suggest an equivalent layering of management functions in the user node to manage these protocols. Figure 2.5 shows the data flow graph of a node so layered as users, the network layer, and the link layer, communicating over a link with a network.

Now that the node has been layered to reflect the X.25 protocol layering, the NETWORK MANAGEMENT FUNCTION together with the PACKETS can be thought of as a virtual machine providing X.25 services to the users. Similarly, the LINK MANAGEMENT FUNCTION, together with the FRAMES can be thought of as a virtual machine providing link services to the X.25 layer.

2.3.3 X.25 PACKET LAYER MANAGER - DATA FLOW

Now that the X.25 protocol management function has been
identified as in figure 2.5, the next stage of design is to
determine the internal data flow of the network management
function, and to allocate functional units to that manager.

The internal organization of this section is inspired
by N. Grootenboer's paper[17], which suggested a dedicated
protocol manager for each virtual connection in use. A pro-
tocol manager is assigned at the time of a connect request
and manages all aspects of the call through to clearing of
the call.

Figure 2.6 shows the data flow of the X.25 Packet Level
Entity. The entity is composed of four main functions, the
management of each Virtual Circuit by a
VIRTUAL_CIRCUIT_MANAGER, the user-entity interface, the allo-
ocation of these VIRTUAL_CIRCUIT_MANAGER's by a DISPATCHER,
and the link interface as handled by ROUTER's. Customers
request a free Virtual Circuit Identifier from a Dispatcher
function (via the interface function function) which has the
responsibility of keeping track of free managers, and assign-
ing them to new calls. Virtual Circuit Managers inform the
Dispatcher of their identifier when free, who passes this to
a waiting user. The user's communications are then directed
to the appropriate VIRTUAL_CIRCUIT_MANAGER to connect or
confirm a Virtual Circuit. All further transactions on the
Virtual Circuit will be managed by the VIRTUAL_CIRCUIT MAN-
AGER, i.e., all SEND's, RECEIVE's, CLEAR's and incoming
packets.

It is unlikely that in any system that users would have
direct access to the virtual circuit manager. If the final
implementation were distributed, the access would be mes-
gages over a link, while if the implementation were common
memory, the interaction would be procedural or controlled by
an operating system. In any case, there will always be an
interface between the user and the manager. The interface
module in this design reflects this premise.
The LINK_MANAGER, by virtue of its one-user protocol, is incapable of sorting or directing frame data to different users in the system. In the X.25_PACKET entity, we have many VIRTUAL_CIRCUIT_MANAGER's and a DISPATCHER, each requiring only packets destined for them. To require them to
access the link for their packets is incompatible with the design of the link protocol, and would seriously impair system performance. They would have to compete with each other to collect packets, without knowing in advance which was theirs.

A transport and routing function is essential here. In this design two transport functions handle this, ROUTER_IN and ROUTER_OUT. The ROUTER_IN function collects INDICATION and CONFIRMATION packets from the LINK_MANAGER and delivers them to the appropriate DISPATCHER or VIRTUAL_CIRCUIT_MANAGER. Similarly, calls to the link on transmission of REQUEST or RESPONSE packets may cause excessive delay in the system. A ROUTER_OUT function is appropriate here, to handle the passage of packets to the link.

2.3.3.1 Virtual Circuit manager - Internal Data flow -

Figure 2.7 shows the internal data flow of the VIRTUAL_CIRCUIT_MANAGER. It is composed of two main data objects, the packets sent on behalf of the user but not yet acknowledged by the network, and packets received from the network, but not yet collected by the user. It also has a VIRTUAL CIRCUIT PROTOCOLCONTROL which manages the over-all operation of the manager. It receives the data or packets, puts them in the appropriate data area until acknowledged or picked up, and passes them on to the link or the user when appropriate. This protocol controller also takes care of retransmissions, circuit initiation and clearing, and reporting error conditions to the DISPATCHER.
FIG 2.7: X.25 VIRTUAL CIRCUIT MANAGER
- INTERNAL DATA FLOW
2.3.3.2 Dispatcher - Internal data Flow -

Figure 2.8 shows in more detail the data flow design of the dispatcher. The dispatcher contains a list of free virtual circuit numbers. When one is requested, either by the interface function on behalf of a user, or by a CONNECT_IND from a remote user, the dispatcher removes a number from its list and passes it to the interface module to access the proper manager for further processing of the call. When VIRTUAL_CIRCUIT_MANAGERS become available, they report their number to the dispatcher to update the list.

FIG 2.8: X.25 DISPATCHER
- INTERNAL DATA FLOW
2.3.4 ALLOCATION OF STRUCTURED OBJECTS

The designer is now in a position to begin allocating system objects to our data flow functions identified.

2.3.4.1 NETWORK MANAGER - EXTERNAL VIEW -

The X.25 service is a conceptual entity which provides services to users. Initially, therefore, we can assign this bubble to a package. *

From this point on we may begin to allocate system structural objects to our entities. Before doing this, we must decide which general method we will use. Piatkowski[13] gives an approach where the system objects and their interactions follow the data flow closely, with data indications and connection indications and responses arriving unsolicited at a user-server interface. An alternate approach is to structure the system so that all interactions between system modules are under the control of the higher level entity. That is, users will place calls on the packet level to send data or receive it, and that the packet level entity will place calls on the link level entity to send and receive data. This method is called a HIERARCHICAL CONTROL APPROACH, and shall be discussed further in the next section. All designs in this chapter shall follow this hierarchical control approach.

*Specifying the X.25 service as a package is a general specification technique, which really does not commit the designer, at this point, to any specific implementation. The underlying entity could be monolithic software, many co-operating tasks, or even hardware. All that is important at this point is that the interface to the package be clean, clear, concise, and complete.
Since we are designing a hierarchical control system to interface with users, the interface to this package will be procedural calls. The interface becomes the procedural interfaces CONNECT_REQUEST, WAIT-ON-CALL, CLEAR, SEND and RECEIVE. The structure diagram for this is shown in figure 2.9 and the formal ADA program specification in 2.10.

2.3.4.2 SYSTEM LAYERING -

Just as the virtual circuit between the end user and the network is managed by the X.25 entity, so the physical link between the end-user and the network is managed by a LINK_MANAGER entity.

Figure 2.9 also shows the LINK MANAGER specified as a package with procedural interfaces SEND and RECEIVE. This is in keeping with the separation of functional managers, and the philosophy of hierarchical, or control oriented access to them.
package X_25_PACKET_LAYER is
  type PORT is limited private;
  procedure PLACE_CALL( ADDRESS: GLOBAL_ADDRESS;
    VIRTUAL_CIRCUIT: out PORT;
    MAX_TIME : SECONDS;
    RESOLT : out R_TYPE);
  procedure CLEAR_CALL( VIRTUAL_CIRCUIT : in out PORT;
    MAX_TIME : SECONDS;
    RESOLT : out R_TYPE);
  procedure WAIT_ON_CALL( SENDER : out GLOBAL_ADDRESS;
    VIRTUAL_CIRCUIT : out PORT;
    MAX_TIME : SECONDS;
    RESOLT : out R_TYPE);
  procedure SEND( VIRTUAL_CIRCUIT : in out PORT;
    MESSAGE : string256;
    MAX_TIME : SECONDS;
    RESOLT : out R_TYPE);
  procedure RECEIVE( VIRTUAL_CIRCUIT : in out PORT;
    MESSAGE : out STRING256;
    MAX_WAIT : SECONDS;
    RESOLT : out R_TYPE);

-- calls made are to
-- send( packet, timeout); --'and
-- receive( packet, timeout -- on each link used.

private
  type PORT is range 0..256;
end X_25_PACKET_LAYER;
package body X_25_PACKET_LAYER is separate;

FIG 2.10: ADA SPECIFICATION OF X.25 PACKAGE INTERFACE
2.3.5 EXAMPLE OF A NETWORK LAYER INTERNAL ORGANIZATION

2.3.5.1 Assignment of Objects

We are now ready to assign ADA objects to the functional entities identified. Since each VIRTUAL_CIRCUIT_MANAGER functions independently from, and concurrently with, the other VIRTUAL_CIRCUIT_MANAGERS and the DISPATCHER, make each a task. For ease of identification and use of these managers, make them an array of tasks, where the virtual circuit number identifies the Virtual Circuit that will be managing the circuit. The DISPATCHER also becomes a task, since it must function concurrently with the VIRTUAL_CIRCUIT_MANAGER's. To function correctly, ROUTER_OUT and ROUTER_IN must also be tasks. Figure 2.11 shows the object diagram of the package, and figure 2.12(a) gives the interface specification for these tasks, while figure 2.12(b) gives the full ADA program specification of the procedural primitives which will access them.
FIGURE 2.11 X.25 PACKET LAYER PACKAGE - INTERNAL STRUCTURE
Package body X.25 PACKET is
  type PORT is range 0 .. 255;
  subtype VALID PORT is PORT range 1 .. 255;
  type CONTROL MESSAGE_KIND is -- Already Declared. Won't repeat here
  type X_25_PACKET(kind: (data, control)) is -- Already declared

task type VIRTUAL_CIRCUIT_MANAGER is -- for one virtual circuit
  entry SEND ( S_MESSAGE : in MESSAGE_TYPE; S_RESULT : out R_TYPE);
  entry RECEIVE(R MESSAGE : out MESSAGE_TYPE; R_RESULT : out R_TYPE);
  entry CLEAR( C RESULT: out R_TYPE);
  entry WAIT ON REPLY( CON RESULT: out R_TYPE);
  entry PUT( R PACKET: in PACKET; P RESULT : out PUT_RESULT_TYPE);
  -- for the transport task(s) to deliver packets
  entry INIT ( NUMBER : integer);
  -- Calls Link.send to send data and control packets
  -- Calls Dispatcher Ready( number ) to receive another user
  -- once the present call cleared
end VIRTUAL_CIRCUIT_MANAGER;

task body VIRTUAL_CIRCUIT_MANAGER is separate;

type CIRCUIT_POOL is ARRAY( VALID PORT ) OF VIRTUAL_CIRCUIT_MANAGER;
VIRTUAL_CIRCUIT: CIRCUIT;

----------------------------------------

task DISPATCHER is
  entry READY( CIRCUIT_ID: in VALID_PORT); -- circuit managers queue here to
  -- wait for next user
  entry PLACE_CALL( ADDRESSEE: in GLOBAL ADDRESS; PORT NUMBER: out VALID PORT; PLACE_RESULT: out R_TYPE);
  entry WAIT ON_CALL( SENDER: out GLOBAL ADDRESS; THIS END, OTHER END: out VALID_PORT_NUMBER);
  entry PUT( REC PACKET: in PACKET_TYPE; P_RESULT : out R_TYPE);
  -- calls ROUTER_OUT. SEND to send to link
  -- connect confirm packets
  -- clear packets
  -- restart packets
end DISPATCHER;

task body DISPATCHER is separate;

----------------------------------------

task type ROUTER_IN_TYPE is
  -- calls LINK RECEIVE to collect packets from the link
  -- calls DISPATCHER.PUT to deposit PLACE_CALL indications
  -- calls VIRTUAL_CIRCUIT( num ). PUT to deposit data packets
  -- one task of this type needed for each link.
end ROUTER_IN_TYPE;

task body ROUTER_IN_TYPE is
  ROUTER IN: ROUTER IN_TYPE; -- -- -- -- -- --
  task ROUTER IN_TYPE is
  entry SEND( LINK: LINK MANAGER; PACKET: X_25_PACKET);
  end ROUTER IN;
end ROUTER IN;
task body ROUTER OUT is separate;

----------------------------------------

FIG 2.12(a)

FIG 2.12(a): ADA PROGRAM SPECIFICATION OF X.25 PACKAGE
- TASK INTERFACES
procedure PLACE_CALL( ADDRESS : GLOBAL_ADDRESS; NUM : out PORT; MAX TIME : in SECONDS; RESULT : R_TYPE);
begin
  select
    DISPATCHER. PLACE_CALL( ADDRESS, CIRCUIT, RESULT)
  or
    VIRTUAL_CIRCUIT( NUM ). MANAGER. WAIT_ON_CALL( ADDRESS, RESULT);
  or
  DELAY( MAX TIME );
  RESULT := TIMEOUT;
    VIRTUAL_CIRCUIT( NUM ). MANAGER. CLEAR( ????????? );
  NUM := 0;
end select;
end PLACE_CALL;

procedure WAIT_ON_CALL( NUM : out VALID PORT; TIME: in SECONDS;
RESULT: in R_TYPE);
begin
  select
    DISPATCHER. WAIT_ON_CALL( NUM, RESULT );
  or
    DELAY( TIME );
  RESULT := TIMEOUT;
end select;
end WAIT_ON_CALL;

procedure SEND( )
begin
  select
    VIRTUAL_CIRCUIT( NUM ). MANAGER. SEND( MESSAGE, RESULT );
  or
    DELAY( MAX TIME );
  NOTIFY_DISPATCHER( NUM );
  RESULT := TIMEOUT;
end select;
exception
  when CONSTRAINT_ERROR => RESULT := ILLEGAL_CIRCUIT;
end SEND;

procedure RECEIVE
  -- SIMILAR TO SEND.

procedure CLEAR( NUM );
begin
  VIRTUAL_CIRCUIT( NUM ). DISPATCHER. CLEAR
exception
  when CONSTRAINT_ERROR => RESULT := ILLEGAL_PORT;
end CLEAR;

FIG 2.12(b)
FIG 2.12(b): ADA PROGRAM SPECIFICATION OF X.25 PACKAGE
- INTERFACE PROCEDURE DETAILS
2.3.5.2 The DISPATCHER -

As mentioned earlier, the DISPATCHER task manages the allocation of virtual circuits and virtual circuit managers. The DISPATCHER's finite state machine is given in figure 2.13 and the ADA program design in figure 2.14.

![Diagram](https://example.com/diagram.png)

**FIG 2.13: DISPATCHER FINITE STATE MACHINE**

Examining the DISPATCHER, we see that it requires entries to permit users to place a call or wait for an incoming call, for VIRTUAL_CIRCUIT_MANAGERS to declare their circuit free, and for ROUTER_IN to deposit CONNECT_INDICATIONS. If no Virtual Circuits are available, users will wait on the PLACE_CALL entry. When both a user is present and a Virtual Circuit is free, the DISPATCHER passes the VIRTUAL_CIRCUIT_MANAGER number to the user, and marks the circuit as busy.
separate (x 25 PACKET_LAYER);
task body DISPATCHER is;
    -- INTERNAL VARIABLES

begin
    -- INITIALIZATION

loop
    select
    when NUMBER_OF_FREE_CIRCUITS # 0 =>
        accept PLACE_CALL (ADDRESS : in GLOBAL_ADDRESS;
            LINE : out PORT;
            RESULT : CONNECT_RESULT) do
            if VALID (GLOBAL_ADDRESS) then -- assume here a package or
                -- data structure which maintains
                -- a list of valid addresses
                    LINE := CIRCUIT_NUMBER;
                    RESULT := OK;
                else
                    RESULT := INVALID_ADDRESS;
                    LINE := 0;
                end if;
            end if;
        end GET_CIRCUIT_NUMBER;
    or
        accept FREE (CIRCUIT_NUMBER : in PORT;
            RESULT : out RESULT_TYPE) do
            FREE_CIRCUITS := FREE_CIRCUITS UNION CIRCUIT_NUMBER;
            RESULT := OK;
        end FREE;
    or
        accept PUT (∑MSG : 25 PACKET; RESULT : PUT_RESULT); case ∑MSG. TYPE IDENTIFIES IS
        when CONNECT REQUEST =>
            if CIRCUIT_NUMBER IN FREE_CIRCUITS then
                if WAIT_ON_CALL. WAITING > 0 then
                    accept WAIT_ON_CALL (FROM : out GLOBAL_ADDRESS;
                        NEW LINE : out PORT;
                        RESULT : out CON_RES) do
                        FROM := ∑MSG. COMING_FROM;
                        RESULT := OK;
                    end WAIT_ON_CALL;
                else
                    BUILD_PACKET (PACKET, CIRCUIT_NUMBER, CLEAR);
                end if;
            else
                BUILD_PACKET (MSG, CIRCUIT_NUMBER, CLEAR);
            end if;
        when others =>; -- Do nothing for now!!!
    end case;
    end select;
exception
    when LINK_DOWN =>
        VIRTUAL_CIRCUIT (CIRCUIT_NUMBER). CLEAR (RESULT);
end loop;
end DISPATCHER;

FIG 2.14: ADA DESIGN SPECIFICATION OF DISPATCHER
If instead, a CONNECT_INDICATION arrives, the DISPATCHER determines that the correct Virtual Circuit is available, and that a user is at WAIT_ON_CALL. It passes the caller's address and the Virtual Circuit Number to the user. Should no circuit or no receiver be available, the dispatcher will send a CLEAR packet to the network on that circuit. Should a call collision occur, that is both the DISPATCHER and network select the same circuit number for different calls, the dispatcher will take no action since it is the networks responsibility under the X.25 protocol to clear the incoming call.

The Virtual Circuit number has now been allocated and is passed to the CONNECT or WAIT_ON_CALL procedure which calls the appropriate VIRTUAL_CIRCUIT_MANAGER and passes it this number on behalf of the user. Once the circuit number has been allocated, the VIRTUAL_CIRCUIT_MANAGER will handle all further transactions of the call.

2.3.5.3 The VIRTUAL_CIRCUIT_MANAGER -

2.3.5.3.1 General -

The Finite state machine for a complete virtual circuit manager is given in figure 2.15. The structure diagram is in figure 2.16, and the ADA program design is presented in figure 2.17.
2.3.5.3.2 Connecting and Clearing Calls

Each VIRTUAL_CIRCUIT_MANAGER has entries to permit a user to send or receive packets, and to clear the call when desired. During the connect phase, the CONNECT procedure will call a CONNECT entry in the VIRTUAL_CIRCUIT_MANAGER on behalf of the user to pass it the address of the desired recipient. The manager will build the CONNECT_REQUEST packet and pass it to ROUTER_OUT for hand-over to the link manager. When a CONNECT_CONFIRM or CLEAR arrives on that circuit, ROUTER_IN passes it to our manager via PUT. The user is then notified of the result and released.
FIG 2.16: VIRTUAL CIRCUIT MANAGER - INTERNAL DESIGN
When an incoming CONNECT_INDICATION packet arrives via ROUTER_IN calling DISPATCHER.PUT, the dispatcher accepts WAIT_ON_CALL, passes the originators address and a Virtual Circuit number to the caller, then releases it. The caller then calls the appropriate VIRTUAL_CIRCUIT_MANAGER CONFIRM to send out the CONNECT-CONFIRM packet.

2.3.5.3.3 Passage of Data -

All of the logic and data structures concerning packet transmission, reception, acknowledgement and retransmission is internal to the VIRTUAL_CIRCUIT_MANAGER. The PROTOCOL CONTROL function identified in figure 2.8 translates into the task algorithm and state control variables. The data passage across the VIRTUAL_CIRCUIT_MANAGER boundaries are handled by the SEND, RECEIVE, CLEAR, CONNECT, and CONFIRM entries. The outstanding packets are stored in two queues, PACKETS_SENT_BUT_NOT_ACKED, and PACKETS_RECEIVED_BUT_NOT_PICKED_UP. The management of these queues is handled by a package of queue management procedures. The primary controlling factors in the task are the implicit state of the task, the size of queues of uncollected packets or unacknowledged packets, and the time in the system of the oldest unacknowledged packet. The manager uses these to determine which entries to open or which paths to follow internally.

In the data transfer state, the VIRTUAL_CIRCUIT_MANAGER will accept SEND if the unacknowledged packet queue is not full, accept RECEIVE if the undelivered packet queue is non-empty, and PUT or CLEAR always. Upon each acceptance of an event, the VIRTUAL_CIRCUIT_MANAGER takes the appropriate action, updating queues and sending packets, then returns to the selective wait for the next event, as shown in figure 2.17.
package Q MANAGER is
  type PACKET_QUEUE is PRIVATE;
  procedure ENQUE( PACKET : access X 25 PACKET;
                   Q : PACKET_QUEUE;
                   NUM : PACKET_COUNTER);
  procedure DEQUE( PACKET : access X 25 PACKET;
                   Q : PACKET_QUEUE;
                   NUM : PACKET_COUNTER);
  procedure DISCARD( Q : PACKET_QUEUE;
                   NUM : PACKET_COUNTER);  
  function FETCH( Q : PACKET_QUEUE;
                   NUM : PACKET_COUNTER) returns access X_25_PACKET;
  function MEMBER( PACKET : access X 25 PACKET;
                   Q : PACKET_QUEUE;
                   NUM : PACKET_COUNTER) returns BOOLEAN;
  function NEW returns access X_25_PACKET;
  function RELEASE( P : X_25_PACKET ) returns X_25_PACKET CONST => null;
private
  type Q_ELEM :
    type PACKET_QUEUE is record
      HEAD, TAIL : access Q_ELEM;
      COUNT : 0..WINDOW;
      end record;
  type Q_ELEM is RECORD
    NEXT : access Q_ELEM;
    DAT : access X_25_PACKET;
    end record;

separate( X_25_PACKET_LAYER );
task body VIRTUAL_CIRCUIT_MANAGER is
  use Q MANAGER;
  with Q MANAGER:
  type CALL_STATES IS ( CLEARED, CONNECTING, CONNECTED, TIMED_OUT ... );
  -- first the circuit identifiers
  CALLEE_ADDRESS : GLOBAL-ADDRESS;
  CIRCUIT : PORT;
  THIS_END, OTHER_END : PORT;
  -- then the control variables
  CALL CONNECTED,
  CLEARED : boolean;
  OK_TO_SEND,
  FLOW_CONTROL : boolean;
  -- the data queues
  MESSAGES_SENT_BUT_NOT_ACKED,
  MESSAGES_RECEIVED_BUT_NOT_PICKED_UP : PACKET_QUEUE;
  -- data transmission variables
  MAX_PACKET_COUNT : integer const := 8;
  SEND_COUNT, RECEIVE_COUNT : 0..MAX_PACKET_COUNT - 1;
  PACKET, JUNK_PACKET : access X_25_PACKET;
  -- ? STATE : CALL_STATES;
  -- Timer and retransmission variables
  TIME_LEFT,
  START_TIME : TIMES;
  NUMBER_OF_RETRANSMISIONS : integer range 0 .. 7;
  CALL_CLEARED : exception;

FIG:2.17(a) ADA SPECIFICATION OF VIRTUAL CIRCUIT MANAGER

QUEUE MANAGEMENT PACKAGE & DECLARATIONS
begin -- internal block to localize exception handling!!
loop
begin
while not CALL_CONNECTED loop
  -- attempt to connect a call, until one is correctly connected
  select
  accept PUT( IN PACKET : in PACKET_TYPE;
             RESULT : out RESULT_TYPE) do RESULT := CLEARED;
          END PUT; -- circuit not yet connected
  or accept SEND( MSG : in STRING;
                RESULT : out RESULT_TYPE) do RESULT := CLEARED;
          end SEND; -- Cannot send until connected;
  or accept RECEIVE( MSG : out STRING;
                    RESULT : out RESULT_TYPE) do
                RESULT := CLEARED;
                MSG := NULL;
          end RECEIVE;
  or accept CLEAR( RESULT : out RESULT_TYPE ) do RESULT := CLEARED;
  or accept CONNECT( CALLEE ADDRESS : in GLOBAL_ADDRESS;
                  CIRCUIT : in PORT;
                  RESULT : out RESULT_TYPE) do
    BUILD PACKET( PACKET, CONNECT-REQUEST, TO_ADDRESS, FROM_ADDRESS
                  CIRCUIT, RESULT );
    TRANSPORT OUT. SEND( PACKET, RESULT );
    if RESULT = LINK_DOWN then raise LINK_DOWN;
    while not CALL_CONNECTED loop
      select
      accept PUT( IN PACKET : in PACKET_TYPE;
                   RESULT : out RESULT_TYPE) do
        case PACKET. KIND is
          when CONNECT-CONFIRM => CALL_CONNECTED := true;
          when CLEAR => RESULT := CLEARED;
          when others => raise CALL CLEARED;
        end case;
        end delay( CONNECT_TIME_OUT ) do RESULT := TIME_OUT;
      end select;
      end loop;
      end CONNECT();
  or accept CONFIRM ( ... ) do
    BUILD PACKET( PACKET, CONNECT-CONFIRM, ... );
    TRANSPORT OUT. SEND( PACKET );
    CALL CONNECTED := true;
    end CONFIRM;
  end select;
end loop;
-- Here we have a connected link!!

FIG 2.17(b): ADA SPECIFICATION OF VIRTUAL CIRCUIT MANAGER
--CONNECTING CALL
while CALL CONNECTED loop;
  TIME_STARTED_WAITING := TIME;
  select
    when MESSAGES SENT BUT NOT ACKED, COUNT < WINDOW =>
      accept SEND(MSG: string256; RESULT: out SEND_RESULT) do
        RESULT := OK;
      end SEND;
      TIME_LEFT := TIME LEFT - TIME + TIME_STARTED_WAITING;
      SEND COUNT := (SEND COUNT + 1) MOD WINDOW;
      BUILD PACKET(PACKET, DATA, CIRCUIT_ID, MSG, SEND COUNT, RECEIVE COUNT);
      TRANSPORT OUT. SEND (PACKET, RESULT);
      ENQUE(PACKET, MESSAGES SENT BUT NOT ACKED);
    or
    when MESSAGES RECEIVED BUT NOT PICKED UP, COUNT > 0 =>
      accept RECEIVE(MSG: string256; RESULT: RECEIVE_RESULT) do
        DEQUE(PACKET, MESSAGES RECEIVED BUT NOT PICKED UP);
        MSG := PACKET. MESSAGE;
        RESULT := OK;
        RELEASE(PACKET);
      end RECEIVE;
      if FLOW CONTROL then
        BUILD PACKET(PACKET, CIRCUIT_ID, RECEIVE READY, RECEIVE COUNT);
        TRANSPORT OUT. SEND(PACKET);
        FLOW CONTROL := false;
      end if;
      RELEASE(PACKET);
    or
    accept CLEAR;
    STATE := CLEARED;
    BUILD PACKET(PACKET, CIRCUIT_ID, CLEAR REQUEST);
    TRANSPORT OUT. SEND(PACKET, RESULT);
  end select;
  case MSG. KIND IS
    when DATA =>
      if FLOW CONTROL then
        -- discard packet
        BUILD PACKET(PACKET, CIRCUIT_ID, RECEIVE NOT READY);
        TRANSPORT OUT. SEND(PACKET, RESULT);
        TIME_LEFT := TIME LEFT - TIME + START TIME;
      elseif not MEMBER(PACKET, MESSAGES SENT BUT NOT ACKED) then
        TIME LEFT := MAX WAITING TIME;
        DISCARD(MESSAGES SENT BUT NOT ACKED, PACKET, RECEIVE COUNT);
        ENQUE(MSG, MESSAGES SENT BUT NOT ACKED);
      if FULL(MESSAGES RECEIVED BUT NOT PICKED UP, WINDOW) then
        BUILD PACKET(PACKET, CIRCUIT_ID, RECEIVE NOT READY);
        TRANSPORT OUT. SEND(PACKET);
        FLOW CONTROL := true;
      end if;
  end case;
end if;

FIG 2.17(c): ADA SPECIFICATION OF VIRTUAL CIRCUIT MANAGER

- DATA TRANSFER - PART 1
when CONTROL =>
    case CONTROL_KIND is
        when REJECT =>
            for I in PACKET, RECEIVE_COUNT..SEND_COUNT loop
                FETCH( PACKET, MESSAGES_SENT_BUT_NOT_DELIVERED);
                TRANSPORT_OUT. SEND( MSG, RESULT);
            end loop;
            PACKET := NEW;
        when CLEAR =>
            CALL_CONNECTED := FALSE;
            BUILD_PACKET( PACKET,
                CLEAR_CONFIRM,
                CIRCUIT_ID);
            TRANSPORT_OUT. SEND('MSG', RESULT);
        when others =>
            end case;
    end case;
    end when;
    end select;
end CIRCUIT_MANAGER;

FIG 2.17(d): ADA SPECIFICATION OF VIRTUAL CIRCUIT MANAGER
- DATA TRANSFER - PART 2
2.3.5.3.4 Retransmission of Packets -

Retransmission of packets can constitute a major headache for system designers in that time-outs are asynchronous events that are difficult to control, and polling to determine time-out is very costly on the system. Instead of using an explicit time task or clock entry, the VIRTUAL-CIRCUIT_MANAGER uses the ADA delay mechanism in the rendezvous. Each packet is stamped with the time of its last transmission. The head of the SENT_BUT_NOT_ACKNOWLEDGED queue is the oldest, hence it will time out and require retransmission first. Before entering the selective accept, the manager determines how much time is left on the head of the queue of unacknowledged packets and sets up a delay parameter accordingly. Should no accept occur before the timer expires, the delay alternative to the accepts is taken, and all unacknowledged packets are retransmitted. If an entry is accepted, the delay is cancelled and re-instituted after the next activity of the manager. This method provides a tight control of timeouts, without having to design an explicit timer mechanism, which is really an implementation detail.

2.3.5.3.5 Clearing the call -

When STATE is set to clear, the manager calls DISPATCHER,FREE to notify it. The manager then performs a selective accept on PUT, CONNECT, or CONFIRM. Any packets arriving at this time via PUT are in error and are just discarded, but ROUTER_IN is released immediately to collect more packets on the link. A call to CONNECT informs the manager that a new connection is being requested, and causes the manager to send a CONNECT_REQUEST packet and wait for the confirmation on PUT, as explained before, while accepting CONFIRM causes a CONNECT_CONFIRM to be sent. Once the circuit is identified as being connected, the manager releases the user and moves to the data transfer part of its algorithm.
2.3.5.4 The Router functions

ROUTER_IN and ROUTER_OUT are standard transport functions for communicating with the link. ROUTER_OUT is absolutely essential as we cannot tolerate more than one task collecting data from the LINK package, and each task must remain active to respond to user requests and time-outs. The ROUTER_OUT task could be replaced by separate entry calls on the LINK package by each VIRTUAL_CIRCUIT_MANAGER, but for conceptual clarity we shall retain the transport task.

2.3.6 SUMMARY AND CONCLUSIONS

This section has presented an internal design of an X.25 packet level service. It is a highly modular design which clearly shows the power and simplicity of structured design. The next section considers some alternate designs and uses them to highlight issues in system design.
2.4 ANOTHER LOOK AT INTERNAL LAYER ARCHITECTURE
- ALTERNATE DESIGNS AND ISSUES

Section 2.1 noted that there were many ways of designing the X.25 package internal structure, once we had decided on the external interface, and that most of these decisions had little impact beyond the scope of the package. This section considers other internal architectures to see how they compare to the one just presented, and to demonstrate design methodologies.

2.4.1 MONOLITHIC DESIGNS

As explained in the previous section, there is a natural layering to the protocols in an X.25 system. Just as one worker in a mailroom could perform all of the necessary tasks, such as collecting mail, sorting it, bagging it, distributing it to appropriate recipients, etc., so also this protocol layering could be handled by one monolithic routine. This monolithic structure would handle both requests from above and frames from over the link. For requests, it would generate the necessary packet, envelope it in a frame, and transmit the frame over the link, and then update the appropriate frame and Virtual Circuit state tables. For incoming frames, it would strip the link control headers and delimiters, remove the packet protocol header, deliver the packet, if data present, and update the appropriate state tables.

In a monolithic structure, the entities mentioned above still exist, but only in a logical sense, in that each entity would comprise the procedures or algorithms in the monolithic structure designed to manage its protocol. A monolithic approach should be avoided for the following reasons:

1. It is very difficult to separate link management
issues from network issues in a monolithic system. For example, the resetting of a link, or the discarding of a frame is a link issue and should not be visible to network logic, unless the error persists or is unrecoverable.

2. The addition of another link, or a new link protocol should never change the way the network views the Link Layer. In a monolithic system, it is difficult to separate the Network and Link Entities to guarantee this modularity.

3. Monolithic software quite thoroughly locks the designer into a rigid system design, normally uniprocessor, which may prove exceedingly costly, especially if the system grows beyond the speed or power of the chosen processor.

4. Each of the protocols could be realized in either software or hardware. A monolithic system obscures the hardware–software trade-offs and reduces the designers' flexibility in choice of system components.

For the reasons mentioned above, all further designs discussed in this section will retain the hierarchical, separate manager approach at the entity level. The implication of this is that, from the outside, or package interface level, the Network manager in each design has the same interface specification.

2.4.2 THE SINGLE MANAGER APPROACH

2.4.2.1 The Design –

Figures 2.18 and 2.19 show the data flow and the object
diagrams of an alternate design of the X.25 service package which performs the same functional service as the initial X.25 design presented. Instead of a dedicated manager for each call, this design creates a single X.25 Protocol Manager Task to manage every virtual circuit. This Manager must maintain an explicit state table for each Virtual Circuit managed, and an explicit data packet counter for that circuit, to maintain the correct sequencing on each circuit. Since the communication link is still full duplex, the manager still requires the services of a router task to listen on the Link Manager to deliver incoming Indications. Since the single manager in this design will now be much busier than any one of the Virtual Circuit Managers in the previous design, the ROUTER_OUT task is far more important to this design.
FIG 2.18: X.25 PACKET LAYER MONOLITHIC MANAGER
- DATA FLOW
Every virtual circuit which is causing a user to wait on an action, be it a receive or wait-on-reply on this managing task, must have a dedicated entry call. That is because the manager has no other way of accessing or servicing a selected user in question. In ADA terms, this is because all entry queues are FIFO, and the designer cannot afford to block one task behind others on a single queue. Since concurrent activities must be mutually non-interfering, any single element in a system accessed concurrently by these activities constitutes a bottleneck. The single element must therefore maintain a separation of the concurrent activities it is servicing. The ADA approach is to allocate separate entries for each separate stream of concurrent tasks. Other systems use alternate approaches such as dedicated mailboxes, priority queuing, etc., to achieve the same end.

2.4.2.2 Issues in design

2.4.2.2.1 The Context-switching Controversy

The design just presented, showed one single task doing the work of the multitude of managers in the previous section. Most proponents of a single task system cite the overhead of context switching as the rationale for not considering a multi-task system such as proposed in the first design. Context switching is only a problem in single processor applications. At this stage, the designer should not have committed himself to any particular underlying hardware architecture, but recognizing that the hardware is often chosen first or forced on the design team arbitrarily, the issue must be considered.
Assume that we have a link that is a serial line, with the data being delivered to the link monitor task, or interrupt service routine, one byte at a time. Since almost all frames will have at least one hundred (100) bytes in them,
and since each frame is passed to the packet layer as a whole, this means that there will be at least 100 task context switches to collect the frame for a single context switch to invoke a task at the packet manager level when the interface is not DMA. This means that even if the introduction of extra tasks doubles or triples the number of context switches at the Packet Manager Level, the actual overhead to the complete system is in the order of 1% or 2%. The conclusion is that context switching additional tasks at this level is not very significant.

If the hardware is DMA, the numbers argument just presented is not so strong, but there are other tradeoffs in the single manager design. A dedicated Virtual Circuit Manager will only be active when there is an incoming or outgoing message on its circuit, while in the single manager design, that manager is active any time there is a service to be performed. The net result in the design first presented, is that there is no extra context switching overhead involved because as soon as the task is active, it is aware of its complete call context, while a single manager will have to perform additional call context switching to get the state the virtual circuit in progress. It is safe to say that the dedicated manager design first presented does not introduce any noticeable extra overhead from the context switching point of view.

A final observation is that there may be extra memory required to hold the context of each task active in a system, with a dedicated task per virtual circuit, if a uniprocessor is chosen for the underlying architecture. In the single manager case, the explicit state tables of the design replace the context storage of the dedicated manager, hence in reality there is little or no difference in the memory requirements of either design.
2.4.2.2.2 Modularity of Design -

When selecting an initial design, there is an inherent conceptual clarity in designing a system so that the component parts are explicitly and visually obvious to all. There is a very high level of complexity in any involved system such as this. It is the responsibility of the designer to manage that complexity and present it in such a way that it can be grasped and digested readily by anyone knowledgable in software systems.

In the first design, it is pictorially obvious to a reader what the general design and composition of the system is. It is not so obvious in the design just presented what the relationship between the various functions of this service are. In the first design, the protocol control can be written as a procedural algorithm, while in the alternate design, state variables and other mechanisms must be incorporated in the design to monitor each circuit independently. In addition, the time-out mechanism will become quite complex since this design will have to check each circuit packet queue for timed out packets.

2.4.2.2.3 Comparison of Number of Entries -

Entries to permit inter-task communication require use of system resources: data structures to queue tasks and requests, context switching time, if applicable, process time for decision-making, the open-ness of entries, etc., message building or parameter passing, etc. It is useful to compare the monolithic manager design to the virtual circuit manager design from the point of view of the entry overhead imposed by the designs.

In the dedicated manager case, we had a single entry for each function: SEND, RECEIVE, CLEAR, PUT, and WAIT_ON_REPLY, in each manager. In the single manager de-
sign, we still have a dedicated entry for each major function on each circuit, in order to preserve the independence of the users and permit effective methods of distinguishing the appropriate user task associated with a given virtual circuit. Queing them all on a single RECEIVE entity, for example, would not allow the manager to release the proper user when a packet arrived for it. As explained earlier, this separation of entries is crucial to the design of concurrent systems. We can therefore only reduce CLEAR and PUT down to a single entry in the manager, since they do not require user task waiting. The conclusion is that the single manager approach does not significantly reduce the number of entries in an entity, or the system overhead created.

2.4.3 FUNCTIONAL MANAGER DESIGN

2.4.3.1 The design -

A second alternative design consists of splitting the single task above into two: one to handle Circuit Allocation and management, and one to handle data transfer, as shown in figure 2.20. This design requires that both tasks must share common data, or must have dedicated entries for sequencing and co-ordination between the Circuit Protocol Controller and the Data Protocol Controller. Note that the comments in the example above, about the equivalence of task context switching in the Dedicated Virtual Circuit Manager design and the state context switching in a Unary Controller also apply here.
FIG 2.20: X.25 PACKET LAYER FUNCTIONAL ENTITY DESIGN
- DATA FLOW
2.4.3.2 Issues -

2.4.3.2.1 Logical Complexity -

The comments in the previous sub-section about the logical complexity being buried in the state transition tables for a design such as this as opposed to the explicit algorithm being written as the task statements, are also valid. An additional constraint to this design is that the data tables must be shared, with explicit updating rules imposed to protect the integrity of the data, or we must co-ordinate the updates by entry calls so that, conceptually, both tasks are updating the sensitive data simultaneously.

2.4.3.2.2 Timeout -

A major consideration in the choice of any protocol design is the time-out mechanism. Section 2.2 showed how a dedicated could handle time-outs implicitly using a DELAY alternative in its selective accepts. In both the single manager design and the functional manager design just presented, each manager will have to manage a time out mechanism for many circuits simultaneously. To follow a similar mechanism as used in the dedicated manager design will require the manager to maintain extra data structures or more intricate logic to allow the one delay alternative to find and time-out the correct packet or circuit. In the single manager case, the complexity is still higher because now all time-outs, either circuit status or data status oriented, must be managed by the same delay alternative.

There are other alternatives to this time-out issue. Two possible ones are:

1. Create a single timer task for the package which will call a TIME entry in each manager to notify
them ( or it ) of the passage of time. Each manager could then maintain and update its own time-out list explicitly in this entry block.

2. Create a dedicated timer task for each packet queued. The task would actually be part of the data structure, hence releasing an acknowledged packet would also terminate its timer. If the timer timed out before acknowledgement, it would call the appropriate entry in the manager to effect retransmission. This method requires many tasks, which appears to oppose the design philosophy of the few tasks in both these latter ( single or functional manager ) designs.

Both of the above designs are more complex than the dedicated manager design of time-out. This is one of the major trade-offs in the selection of any design and should taken into serious consideration.

2.4.4 DYNAMIC MANAGER DESIGN

2.4.4.1 The Design —

Another design alternative is to use the design as originally presented, but allow the dispatcher to create the Virtual Circuit Manager tasks needed, and permit them to exist only for the existence of the call. The user would here be passed an access variable to the task, and be allowed to call it this way.
2.4.4.2 Issues -

2.4.4.2.1 Fixed task rationale -

There has not been to this point, an explanation for the use of an array of fixed tasks as VIRTUAL_CIRCUIT_MANAGERS in the first design. In any X.25 server, there will be an upper limit on the number of virtual circuits that the user will be permitted to simultaneously use. This limit is assigned by the network and is a function of the class of service contracted for. It is quite natural to create one Virtual Circuit Manager for each of the Circuits possible. These Virtual Circuits, hence the managers, are a scarce resource, and there must be a dispatching function which assigns a free Virtual Circuit Manager to a user, to maintain order in the system. By having the VIRTUAL_CIRCUIT_MANAGER handle the circuit of the same number, we have eliminated most of the mapping and look-up tables, or dynamic space allocation necessary in other designs, and at the same time enhanced conceptual clarity.

In this Dynamic Manager design, to enable the ROUTER_IN task to access the VIRTUAL_CIRCUIT_MANAGER, we require a look-up table in this design. This table will be maintained by the DISPATCHER which writes the access variable of the newly created VIRTUAL_CIRCUIT_MANAGER into the appropriate array location in the table, and used by the ROUTER_IN TASK to access the manager handling the circuit named in the incoming packet. Since we already had a table for the DISPATCHER to maintain a record of free and in-use managers, we can allow this table to be placed external to the DISPATCHER and to contain the access variable of each VIRTUAL_CIRCUIT_MANAGER as well.

When a task finishes, the dispatcher must know that the circuit is free, and that it can create a new task to manage that circuit when it is next allocated, hence the
VIRTUAL_CIRCUIT_MANAGER must still call DISPATCHER.FREE to notify the DISPATCHER when its circuit is cleared or in difficulty. The VIRTUAL_CIRCUIT_MANAGER need not carry its own access variable since by returning the circuit number to DISPATCHER.FREE will permit the DISPATCHER to update the table.

2.4.4.2.2 Exceptions -

Another consideration in this design is to prevent the ROUTER_IN function from attempting to access a terminated task. A check of the VIRTUAL_CIRCUIT_TO_MANAGER table on each access will usually suffice, however there is a scenario which could lock up the system. Should ROUTER_IN obtain the access variable to a VIRTUAL_CIRCUIT_MANAGER task in between the time the manager declares its circuit clear to the DISPATCHER and terminates, and the time when the DISPATCHER updates the table, the ROUTER_IN TASK would be accessing a terminated task.

To clear this condition, we must take one of the following actions in the design:

1. Design the VIRTUAL_CIRCUIT_MANAGER so that after it declares its circuit free, it performs a selective wait on all of its entries, together with a delay option. Any entry calls now issued on the manager will be cause the manager to return a CLEARED status to the caller.

2. Include a local exception handler the design of each task or primitive rendezvousing with a manager to recover from this erroneous condition, should it occur.

Other methods, such as creating a dedicated manager to manage the table and pass access variables to callers show lit-
tle promise, since the basic erroneous condition still exists. The second method appears to be the cleanest and simplest method; that is, let the error occur, then recover gracefully in an exception handler.

Note that the initial dedicated manager design did not have this difficulty, since each manager remained active and could report erroneous calls to users as they occurred.

2.4.5 CONCLUSIONS

Three alternate designs were examined in this section, the single manager, the dual manager, and the dynamic, dedicated VIRTUAL_CIRCUIT_MANAGER. Each design solved some areas that created difficulties in other designs, but also created new problems. The most important lesson from these comparisons is that whatever design is selected, it should be for simplicity and ease of comprehension by the human designers and implementers. Other issues can be traded off, but this one remains critical.

2.5 EXTENDING THE LAYER CONCEPT

To this point in this chapter, we have been discussing design methodologies in the context of the X.25 Packet Switched protocol. This section examines briefly the design of nodes to provide services in a more sophisticated layered protocol, the Open System Interconnection of the International Organization for Standards.

Since the introduction of X.25, the International Organization for Standards has been developing a protocol standard to provide many services to users over a distribut-
ed environment. Some of the services envisioned are File Transfer, Data Encryption, Electronic Mail and Virtual Terminal support over one or more distributed systems. These services must hide the idiosyncrasies of the underlying system as well as manage the interconnected system on behalf of the user.

Like X.25, the Open System Interconnection that the ISO has developed is based on the concept of individual ENTITIES within a node providing services to users or higher level ENTITIES. To provide the service, ENTITIES must interact with remote ENTITIES. The ENTITIES interact by exchanging PROTOCOL DATA UNITS via services provided by lower level ENTITIES.

Although ENTITIES may rely upon other ENTITIES to manage the exchange of PROTOCOL DATA UNITS, each entity is independent from the others, hence the PROTOCOL DATA UNITS exchanged are independent. To permit these PROTOCOL DATA UNITS to share the same lower levels, the data units are nested with the higher level units nested immediately inside of the PROTOCOL DATA UNIT of the entity it called, exactly the way that the X.25 NETWORK LEVEL PROTOCOL DATA UNIT was nested inside the LINK PROTOCOL DATA UNIT, or FRAME.

2.5.1 FRAME-WORK OF OPEN SYSTEM INTERCONNECTION

Just as the X.25 specification only specified the PROTOCOL DATA UNITS exchanged, plus node action upon receipt of each kind of data unit, so also the Open System Interconnection specifies only the PROTOCOL DATA UNITS exchanged between peer entities and the associated state transitions of each entity. The Open System Interconnection Specification does go further, however, in specifying a general layering of entities within a node in the system, and a conceptual relationship between interacting entities, that is SERVICE ACCESS POINTS for an ENTITY placing service requests.
on a lower level entity, and N-CONNECTIONS, and N-CONNECTION END POINTS for the logical connection between peer N-ENTITIES, that is, ENTITIES in level N.

As stated above, the Open System Interconnection specification identifies layers within the system. There are seven layers identified altogether: a PHYSICAL, LINK, and NETWORK level as discussed in X.25, a TRANSPORT level to provide end-to-end services and longer messages than the Network provides, a SESSION level to provide checkpointing and restarting logic, a PRESENTATION level to manage data encryption, etc., and finally an APPLICATION level for high level applications such as electronic mail.

Although the Open System Interconnection specification dictates the layering, etc., as shown above, it does not address any other design or implementation details. The design of a node to meet the document produced by the International Organization for Standards is again the perview of the System Designer.

2.5.2 SOME PRINCIPLES FOR OSI SYSTEM DESIGN

As stated before, there is no intent to attempt to design a node for OSI here. It is worthwhile, however, to show how the design techniques we have developed apply.

We now have recipe for a design of such a system. The general points are as follows:

1. Examine the data flow of an each entity, external, between peers, and internal.

2. Make each OSI layer a package.

3. Make each entity within the layer a visible package.
4. Define the services to be provided by each entity. Make the service primitives Procedure Calls to the package.

5. Define and Code the Protocol Data Units.

6. Assign objects, tasks, packages, data structures, to the "bubbles" in the data flow.

7. Code the design.

This general recipe for the design of an OSI system came directly from our experience with the X.25 design. The design of many other systems can also follow the same guiding principles.
CHAPTER 3
FROM THE ISO FDT TO SYSTEM DESIGN

3.1 INTRODUCTION

3.1.1 Overview of the ISO proposed Formal Description Technique

The International Organization for Standards Ad Hoc Working Group on Formal Description Techniques is proposing a Formal Description Technique (FDT) based on extensions to the Pascal programming language [10]. This Formal Description Technique is used to specify:

1. the various possible states that an entity can be in;

2. the external events that can cause the state transitions;

3. the relationship between an entity and those communicating with it; and,

4. the format of primitives received and transmitted by the entity.

The proposed FDT adds to the Pascal language the concept of a MODULE, its major states and TRANSITIONS between those states, and the INTERACTION of the MODULE with its environment over CHANNEL’s. The principles are contained in the ISO ad hoc group on FDT, subgroup B’s report to the ISO[10]. Figure 3.1 shows Buhr’s concept of two MODULE’s
connected by a channel, or INTERACTION, with state diagrams included. Figure 3.2 sketches the general relationship between this FDT and Buhr's Data Flow graphs[19].

![Diagram of State Diagram, Module, and Interactions]

**FIGURE 3.1**
PICTORIAL REPRESENTATION OF FORMAL DESCRIPTION TECHNIQUE - MODULES AND INTERACTIONS

The original purpose of this chapter was to show how the FDT specification as proposed by Bochmann[3] could be translated into a Structured Design and ADA specification of OSI. Studies showed, however, that the FDT is really closer to the Data Flow design level. This chapter, therefore, shows how the proposed FDT relates to our design process.

This chapter, shows the method by example, first by summarising a simple example by G. V. Bochmann[2(Annex 1)], then by using the FDT as the first stage in designing Transport Entities. The actual designs will be expressed in
structure diagrams and specified in ADA, following the method of the previous chapter. Through this vehicle, the relationship of the proposed OSI FDT to Buhr's Data Flow Graphs, and to design, shall become apparent.

\[ \text{FIGURE 3.2} \]
\text{MAPPING FDT TO DATA FLOW GRAPHS}

3.1.2 PLACE OF FORMAL DESCRIPTION IN SYSTEM DESIGN

In specifying a system, a formal description specification should specify only the essential elements in the system that a final design must adhere to. It should not imply
or force a specific design and the final design must be verifiable in terms of the specification. It belongs at the beginning of the design process, before the "Bubble Graph" stage of Buhr. Through the use of the FDT, this chapter will show that the proposed FDT is really replacing the "Bubble Graph" stage of design, and that by its nature, it strongly implies a final design.

3.2 USING THE ISO FORMAL DESCRIPTION TECHNIQUE:
BOCHMANN’S CLASS O TRANSPORT MANAGER

3.2.1 The OSI FORMAL DESCRIPTION TECHNIQUE

This section will summarize Bochmann's[2(Annex 1)] description of a Single Connection Class O Transport Entity to show how the FDT is used, and to prepare the way for more detailed examples to follow.

In the first specification of a Transport Entity, Bochmann presents an entity capable of managing a single Class O Connection to a single Service Access Point (SAP). A pictorial representation of this design is given in figure 3.3, and a summary of Bochmann's formal description is in figure 3.4(a) and (b).

As shown in figure 3.3, the entity is comprised of three modules, a PROTOCOL MANAGER module which provides services to the Session Layer and interacts with the other two modules, a TPDU Interface and Control module which builds and strips Protocol Data Units and routes PDU's to the Network Layer for the manager, and a Buffer manager which buffers Service Data Units on behalf of the manager. The formal description of the PROTOCOL MANAGER is the only one presented by Bochmann, hence the state diagram of the PROTOCOL MANAGER is the only one shown in 3.3.
The PROTOCOL_MANAGER interacts with the session layer through a Transport Service Access Point, an INTERACTION called TSAP, with the ROUTER via the INTERACTION called MAPPER, and with the BUFFER_MANAGER via OUT_BUFFER and IN_BUFFER. The PROTOCOL_MANAGER has four major states, given by the variable STATE, of IDLE, WAIT_FOR_CC, WAIT_FOR_TCONNECT_RESP, and DATA_TRANSFER. Primitives initiated by modules connected by an interaction cause a state transition to occur in this module, and corresponding events occur. For example, when the session entity initiates...
TSAP.T_CONNECT_REQ and PROTOCOL_MANAGER is in the IDLE state, the statement "when TSAP.T_CONNECT_REQ( ... )" causes the appropriate state transition TO WAIT_FOR_CC, PROVIDING certain conditions are met. Similarly, all the other states, the possible activating interactions, and the appropriate transitions are specified by this FDT for the PROTOCOL_MANAGER.

Other modules which comprise the TRANSPORT ENTITY are similarly specified, although their description is not included here, or by Bochmann.
TRANSPORT SERVICE SPECIFICATION

The following modules constitute a transport service specification for a transport entity providing a single class 0 service to one
Transport Service Access Point.

- TRANSPORT_PROTOCOL_MANAGER
- LOCAL_BUFFER_MANAGER
- TPDU_ROUTER_AND_CONSTRUCTOR

type
T_address_type = ...;
TCEP_identifier_type = ...;
quality_of_service = ...;

interaction
TS_access_point( TS_user, TS_server ) is
  by user:
    T_CONNECT_req( TCEPI : TCEP_identifier_type;
      TO_T_ADDRESS : T_address_type;
      QOTS_request : Quality_of_TS_type;
      options : option_type;
      TS_connect_data : TS_connect_data_type);
    T_CONNECT_response( ... );
    T_DISCONNECT_req( ... );
    T_DATA_req( ... );
    T_EXPEDITEDATA_req( ... );
  by server:
    T_CONNECT_ind( ... );
    T_CONNECT_confirmin( ... );
    T_DISCONNECT_req( ... );
    T_DATA_ind( ... );
    T_EXPEDITEDATA_ind( ... );

type
  ... (other types, such as disconnect_reasons, reject_reasons,
    calling and called addresses, etc.)

interaction
TPDU_and_control( manager, NS_mapper ); (-- builds PDU's for manager)
  by manager, mapper:
    CR( credit: ... ; source_ref: ... ; class: ... ; options: ... ;
        calling_address, called_address: ... ; max_size: ... ; ... );
    CC( ... ); ( connect confirm )
    DR( ... ); ( disconnect request )
    DT( ... ); ( data )
    ERR( ... ); ( error )
  by manager:
    N_DISCONNECT_req( NCEP_id : ... ; reason : ... );
  by mapper: ...

interaction
local_buffer( user, buffer ) is
  by user: clear;
    set-max-size( ... );
    append( ... );
  by buffer: get_next( ... );

FIGURE 3.4(a)
PARTIAL FORMAL DESCRIPTION - SINGLE CONNECTION TRANSPORT ENTITY
- INTERACTIONS (DUE TO BOCHMANN)
module TRANSPORT_manager( TSAP: TS_access_point( TS_server );
   mapping: TPDU_and_control( manager );
  out_buffer, in_buffer, local_buffer;
VAR
  state : ( idle, wait_for_CC, wait_for_T_CONNECT_resp, data_transfer);
( other variables )
initializations
  state := idle;
...
(transitions)
  from idle
  when TSAP, T_CONNECT_req( ... )
    provided ... ( can provide service requested )
    to wait_for_CC
    begin
      local_reference := ...
      TPDU_size := ...
      to_address := ...
    end;
  provided ... ( cannot provide service )
    to idle
    begin
      TSAP, DISCONNECT_ind( TCEPI, reason );
    when mapping, CR( ... );
    provided ... ( can provide connection )
    to wait_for_T_CONNECT_response begin
      TSAP, T_CONNECT_ind( ... );
    end;
    provided ... ( cannot provide connection ) begin
      ...
    mapping, DR( ... );
  end;
  from wait_for_CC
  when mapping, CC( ... );
    to data_transfer begin ... end;
  when mapping, DR( ... );
    to idle ...
...
end TS_MANAGER;

module BUFFER_MANAGER( buffer: ... ) is ... end BUFFER-MANAGER;
module TPDU_ROUTER( mapping: ... ; NSAP: ... ); is ... end TPDU_ROUTER;

FIGURE 3.4(b)
SKELETON FORMAL DESCRIPTION - SINGLE CONNECTION TRANSPORT ENTITY
- TRANSITIONS (DUE TO BOCHMANN)

3.2.2 Discussion

Figure 3.2 and figure 3.3, the view of the entity presented in 3.3 and specified in 3.4(a&b) are reminiscent of
the data - flow design approach of chapter 2, where the basic functional units of the entity and the data exchanged were first identified. This closeness to the data flow design approach suggests that it might unavoidably imply a system organization, just as did the data flow designs of the previous chapter. The rest of this chapter will examine this supposition by investigating alternate extensions to this simple specification of a Transport Entity and deriving corresponding system designs.

3.3 USING THE ISO FDT: A SECOND EXAMPLE

This section investigates one possible design of a multi-connection Transport entity. It is inspired by the FDT of 3.1 and the DISPATCHER - VIRTUAL_CIRCUIT_MANAGER DESIGN of chapter 2. It is not Bochmann’s approach to the general design[2, Annex 2], which shall be discussed in the next section, but it does follow from his single connection manager example.

While the approach to the interactions of a manager to its environment in Bochmann’s single manager example is similar to the X.25 managers in Chapter 2, there are differences. One example is: in Chapter 2, the ROUTER_IN task passed complete packets to the manager, while in Bochmann’s example, the ROUTER built and stripped PDU’s. Such differences have been resolved in favour of Bochmann’s approach.

This example will manage multiple connections concurrently by multiplying the PROTOCOL_MANAGER module of 3.1. To aid the managers in this task, additional modules are needed to interface the TSAP interactions with the correct
manager, and to access multiple network connections.

3.3.1 DATA FLOW GRAPH INSPIRED DEVELOPMENT OF A SKELETON FDT

Figure 3.5 shows a multiple connection transport entity interacting with a session entity over a single TSAP. Each TRANSPORT_CONNECTION is managed by a unique PROTOCOL_MANAGER module which is specified by an FDT module with its own state transition model, following the specification of the CONNECTION_MANAGER module in 3.1. This manager will handle all phases of a connection from connect request to disconnect, as in the previous section. The only difference is that CONNECT_IND PDU's will not be routed to a PROTOCOL_MANAGER, since a connection has not yet been assigned. Clearly a DISPATCHER module is required to field CONNECT_IND's and to assign CONNECTION MANAGER's as needed.
Since each interaction is occurring over one channel, as before in 3.1, an INTERFACE module is needed. This module maps the interaction received over the TSAP to the manager, and initiates a similar primitive over its channel to the correct PROTOCOL_MANAGER. A diagram of the INTERFACE module is given in figure 3.6, and its FDT skeleton specification is given in 3.8. All CONNECT_REQ's are passed first to the DISPATCHER where correct addressing, availability of a connection and availability of a buffer are verified, and the proper CONNECTION_ID and CONNECTION_MANAGER are returned. The updated CONNECT_REQ is then passed to the manager for further connection. All further interactions on this T_CONNECTION will have the CONNECTION_MANAGER identified, hence will be automatically routed to that manager for processing.
Building, stripping, and routing PDU's to the appropriate network connection is the function of the ROUTER_IN and ROUTER_OUT modules, one for each network connection. The ROUTER_OUT puts Service Data units and session PDU's into transport PDU's and passes them to the appropriate network connection for packetizing. The ROUTER_IN module collects packets from a network connection, builds transport PDU's, and when a PDU is complete, passes it to the appropriate manager.

Each router pair also needs a ROUTER_CONTROLLER to provide the correct assignment of router modules and to pass the identification of the assigned router tasks to the CONNECTION_MANAGER. Figure 3.7 gives a skeleton FDT specifica-
tion of the various channels used by these co-operating modules. There is no FDT specification of these modules given, but their design and specification follows from the previous discussions.

The BUFFER_MANAGER, as before, will allocate space for each PROTOCOL MANAGER, and will queue each manager's data, and prompt it if there is a time-out.
TRANSPORT SERVICE SPECIFICATION - Multi-Connection Service Provider

(The following modules constitute a transport service specification for a transport entity providing a single class 0 service to one transport service access point.
- TSAP INTERFACE MODULE
- DISPATCHER
- TRANSPORT PROTOCOL MANAGER
- LOCAL BUFFER MANAGER
- TPD U ROUTER AND CONSTRUCTOR)

type address_type = ...
TCEP_identifier_type = ...
quality_of_service = ...

interaction
TS_access_point( TS_user, TS_server ) is
- (This interaction used by TSAP INTERFACE MODULE to provide a service interface to a user entity)

by user:
T_CONNECT_req( TCEP, T_CONNECT_req_data );
to T_ADDRESS,
fr T_ADDRESS : T_address_type;
QOTS_request : Quality_of_TS_type;
options : option_type;
TS_connect_data : TS_connect_data_type);

T_DISCONNECT_response(...);
T_DATA_req(...);
T_EXPEDITEDATA_req(...);

by server:
T_CONNECT_ind(...);
T_CONNECT_confirm(...);
T_DISCONNECT_req(...);
T_DATA_ind(...);
T_EXPEDITEDATA_ind(...);

interaction
allocate_result = (OK, no_local_buffers, bad_to_address,
bad_from_address, unable_to_provide_service);

connection_request( interface, dispatcher ) is
- (by interface)

T_manager_allocate_req( to T_address,
from T_address : T_address_type;
QOTS req ; quality_of_TS_type);

(by dispatcher)
T_manager_allocate_conf( to T_address,
from T_address : T_address_type;
manager_allocated : connection_manager_type;
result : allocate_result);

T_connect_ind(...); (-- here the CEP identifier is also the manager identifier)

(interaction)
connection_control( interface, manager ) is the same as TS_access_point
( where the interface module is the user now.)

FIGURE 3.7(a)
TRANSPORT ENTITY FDT SPECIFICATION - INTERACTIONS ( PT 1)
interaction
manager_control( dispatcher, manager ) is
  by manager:
    free( my_id : manager_number );
    by dispatcher:
      TS_disconnect_req( ... ); ( -- to disallow unconnectable calls )
  by dispatcher:
    TE_disconnect_req( ... );

interaction
dispatcher_access( dispatcher, NS_mapper );
  by manager:
    connection_update( connection_id : TCEP_id_type;
                       manager : manager_number;
                       new_condition : ( assign, disassign ) );
  by NS_mapper:
    T_connect_ind( ... ); ( -- dispatcher sees all connect requests
                             and indications for validation )


type ...
    ( other types, such as disconnect_reasons, reject_reasons,
      calling and called addresses, etc. )

interaction
TPDU_and_control( manager, NS_mapper ); ( -- builds PDU's for manager )
  by manager, mapper:
    CR( ... ); { connect confirm }
    DT( ... ); { data }
    ERR( ... ); { error }
  by manager:
    N_DISCONNECT_req( NCEP_id : ... ; reason : ... );
    CR( credit: ... ; source_ref: ... ; class: ... ; options: ... ;
                   calling_address, called_address: ... ; max_size: ... ; ... ) ;

    by mapper:
      ...

interaction
local_buffer( user, buffer ) is
  by user: clear;
    set-max_size( ... );
    append( ... );
  by buffer: get_next( ... );
module TSAP_INTERFACE(
    TSAP: array(-1..number SAPs) of TS_ACCESS_POINT( server);
    Manager_connect: array(1..number_of_managers) of TS_ACCESS_POINT( user);
    Dispatcher_connect: connection_request);

VAR
    state: ( dummy );

TRANSITIONS
    from dummy to dummy:
        any TSAP_id: TSAP_id_type with TSAP( TSAP_id ) do
            when T_connect_req do
                dispatcher_connect. connect_req( ... );
            end;
            when dispatcher_connect. allocate_manager( ... , manager_number, ... ) do
                manager_connect( manager_number ). T_connect_req( ... );
            end;
        end;
        any TSAP_id: TSAP_id_type
            with TSAP( TSAP_id ) do
            when T_connect_conf( ... ) do
                manager_connect( manager_num( TCEPI ) ). T_connect_conf( ... );
            end;
        end;
        ...
        any manager_number: manager-number_type
            with manager_connect( manager_number ) do
            when T_connect_ind( ... ) do
                TSAP( TSAP_of_manager( manager_number ) ). T_connect_ind;
            end;
        end;
        ...

module DISPATCHER( DISPATCHER_CONNECTION; connection_request;
    CONTROL : array[CEP_id] of manager_control;
    connection : dispatcher_access ) is

VAR
    state : ( idle );
    free_managers : set of TCEPI_id;

TRANSITIONS
    from idle to idle; (-- this module only routes requests )
        any CEP : TCEPI_id
            when control( CEP ), free do begin
                free_managers := free_managers + CEP;
            end;
            when DISPATCHER_CONNECTION. CONNECT_REQ( ... ) do begin
                ( -- verify addresses, free space credit )
                manager := select-from( free_managers );
                if manager # NIL then DISPATCHER_CONNECTION. T_MANAGER_ALLOCATE_CONV( ... , manager, ... )
            else DISPATCHER_CONNECTION. T_DISCONNECT_REQ( ... , reason );
            end;
            when connection. CONNECT_REQ( ... ) do begin
                ( verify local buffer credits, addresses )
                if manager # NIL and buffer allocated and addresses_ok then
                    DISPATCHER_CONNECTION. T_MANAGER_ALLOCATE( ... )
                else CONNECTION. DR( ... , reason );
            end;
        end; ( dispatcher )

FIGURE 3.8(a)
TRANSPORT ENTITY FDT SPECIFICATION
    - INTERFACE & DISPATCHER
module TRANSPORT_manager( TSAP: TS_access_point( TS_server ));
        mapping: TPDU and control( manager );
        out_buffer, in_buffer : local_buffer;

VAR
    state : ( idle, wait_for_CC, wait_for_T_CONNECT_resp, data_transfer );

    ( other variables )

initializations
    state := idle;

    ...

(transitions)
from idle
    when TSAP. T_CONNECT_req( ... )
        provided ... ( can provide service requested )
            to wait_for_CC
        begin
            local_reference := ...;
            TPDU_size := ...;
            to_address := ...;
            ...
        end;
        provided ... ( cannot provide service )
            to idle
        begin
            TSAP. DISCONNECT_ind( TCEPI, reason );
            when mapping. DR( ... );
            provided ... ( can provide connection )
                to wait_for_T_CONNECT_response
            begin
                ...
                TSAP. T_CONNECT_ind( ... );
            end;
            provided ... ( cannot provide connection ) begin
                ...
                mapping. DR( ... );
            end;
    from wait_for_CC
        when mapping. CC( ... );
        to data_transfer begin ... end;
        when mapping. DR( ... );
        to idle ...;
    ...
end TRANSPORT_MANAGER;

module ROUTER_OUT( ... ) is ... end ROUTER_OUT;
module ROUTER_IN( ... ) is ... end ROUTER_IN;
module ROUTER_MANAGER( ... ) is ... end ROUTER_MANAGER;

FIGURE 3.8(b)
TRANSPORT ENTITY FDT SPECIFICATION
- TRANSPORT_MANAGER and ROUTER's

3.3.2 ADA DESIGN OF MULTI-CONNECTION TRANSPORT ENTITY

The design of the transport entity follows almost directly from the FDT specification in the preceding section.
As in chapter 2, design begins with a Data Flow Graph, followed by a Structure Graph and finally the ADA Specification.

3.3.2.1 DATA FLOW -

The Data Flow Graph of the Multi-connection Transport Entity follows directly from the FDT diagrams and is shown in figure 3.10. Each module is replaced with a "bubble" functional entity, and each channel is replaced with the appropriate data flow paths representing the direction of each interaction on a channel. There is a difference in this notation, however, in that bubbles can be nested inside other bubbles to reflect the relative interactions. Here, the various functional bubbles of the entity are shown inside the "ENTITY BUBBLE" which is in the "TRANSPORT LAYER BUBBLE". In the FDT notation, there is no other way except comments to denote the extent of structures larger than a module. As in the FDT diagrams, the state transitions are included in each bubble.

3.3.2.2 STRUCTURED DIAGRAMS -

The second stage in the design is the assignment of ADA structures to the data flow bubbles. This is done here following the method of chapter 2. Figure 3.10 shows the structure graph design. The TRANSPORT ENTITY shall become a package, with all accesses to this package from above being procedure calls. The procedure calls then become the target of the interface bubble under the mapping. The TCEP_IDENTIFIER for each connection then becomes a pair of parameters, the SAP number and the local connection number. In the final ADA design, these will become limited private data.
Transport Layer Multiple Class 0 Connection Entity - Data Flow Graph
Maintaining the hierarchical concept, all packets from below must be collected on behalf of the entity by TRANSPORT or ROUTING tasks, as was done in chapter 2, so the ROUTER modules of the FDT design become at least a collection of tasks, each monitoring one network connection.

As was shown in the data flow graph, each connection manager is an independent, concurrent function. Following the guidelines of chapter 2, each becomes a separate task, only in this design, they are dynamic tasks to permit a more flexible use of resources.

A single CONNECTION_MANAGER will have a separate entry for each possible interaction it services, - CONNECT_REQUEST, CONNECT_CONFIRM, DISCONNECT_REQ, DATA_REQUEST, DATA_IND, and EX_DATA_REQ. Expedited data indications will be passed at DATA_IND, and disconnect indications will be delivered to the user at the first entry accepted. PDU's from below will, in this design, be stripped by the ROUTER_IN task. Instead of calling one entry PUT, as was done in chapter 2, this CONNECTION_MANAGER will follow Bochmann's design and have one entry for each type of PDU recognized. The ROUTER_IN task must determine the PDU type and call the appropriate entry. The advantage of this method is that the manager need only accept the PDU's which are valid in the context of its current state.

*In this case, should some or all of the manager tasks share memory, then the termination of managers of disconnected connections frees considerable memory, permitting the acceptance of calls needing credit for more space than was previously available.
Fig 3.10: Transport Layer Multiple Class 0 Connection Entity - Structure Graph Representation
As in the X.25 design of chapter 2, a DISPATCHER task is needed to assign CONNECTION MANAGERS to valid connections. It will receive requests from the PLACE_CALL procedure, and validate addresses, etc., then create and assign a manager to that connection. Similarly, all TPDU_CONNECT_INDICATIONS will come to this manager for validation. Allowable connections will have a manager assigned, and the CONNECT_IND information passed to a waiting task at CONNECT_IND. The DISPATCHER will be aware of terminated connections via the FREE entry. CONNECTION MANAGER's will be responsible for terminating themselves after notifying the DISPATCHER via FREE and ensuring all callers on its entries are notified and released.

The only portion left in this phase of the design is the ROUTER_OUT. Since ROUTER_IN is stripping PDU's, and Bochmann's design had the router function also building PDU's, it is reasonable to have the routing function do the same. The ROUTER_IN tasks cannot do this, since they are normally suspended on network connection. A ROUTER_OUT task is therefore necessary to build and send the PDU's. Since the network may impose flow control, one ROUTER_OUT task is needed for each network connection.
TRANSPORT SERVICE SPECIFICATION - Multi-Connection Service Provider

ADA SPECIFICATION

package TS_ENTITY is
  type T_address_type = ...;
  type TRANSPORT_ADDRESS = ...;
  type QUALITY_OF_SERVICE = ...;
  type CEPI is limited private;
  ...
  procedure T_CONNECT_req( TCEPI : TCEP_identifier_type;
      to_T_ADDRESS,
      fr_T_ADDRESS : in T_address_type;
      QOTS_request : in Quality_of_TS_type;
      options : in option_type;
      TS_connect_data : in TS_connect_data_type);
  procedure T_CONNECT_response( ... );
  procedure T_CONNECT_ind( ... ); -- expedited data indications also
  procedure T_CONNECT_confirm( ... );
  procedure T_DISCONNECT_req( ... );
  procedure T_DATA req( ... );
  procedure T_EXPEDITED_DATA req(...);
  procedure T_DATA_ind( ... ); -- indications also return disconnect req.
  private;
    type CLASS_0_MANAGER is; -- incomplete task specification
    type MANAGER_PTR is access CLASS_0_MANAGER;
    type TSAP ID TYPE is ...
    type T_CONNECTION_ID TYPE is ...;
    type TCEP_identifier_type is record
      TSAP ID: TSAP ID TYPE;
      CONNECTION: T_CONNECTION_ID TYPE;
      MANAGER : access CLASS_0_MANAGER;
    end
;
end TS_ENTITY;

FIGURE 3.11
ADA INTERFACE SPECIFICATION OF TRANSPORT ENTITY

3.3.2.3 ADA SPECIFICATION -

Figures 3.11 to 3.14 show the ADA specification of the multiple connection TRANSPORT_LAYER_ENTITY. Figure 3.11 gives the interface specification of the TS_ENTITY package and its procedure calls T_CONNECT_REQ, etc. Figure 3.12(a) gives the interface specification of the DISPATCHER task and of the CLASS_0_MANAGER tasks in this package, while 3.12(b) gives the skeleton interfaces of the ROUTER tasks and the skeletons of the package interface procedures. In these latter procedures, the code is straightforward, and comes almost directly from the X.25 example previously, but using naming conventions driven by the FDT specification.

Figure 3.13 gives the internal skeleton specification
of the DISPATCHER task, while figure 3.14 gives the internal skeleton specification of the CLASS_0_MANAGER. In these tasks there is no explicit state variable for the task to use: rather the <selective> ACCEPT statements govern the state of the task. For example, the CLASS_0_MANAGER waits initially on the entries CONNECT_REQ or GIVE_CONNECT_INFO. This is comparable to the IDLE state. Acceptance of one of these lets the task proceed to suspending on TPDU_CONNECT_CONF, TPDU_DISCONNECT_CONF, etc., or on CONNECT_CONF, DISCONNECT_CONF, etc., which is comparable to the WAIT_FOR_CC or WAIT_FOR_DISCONNECT_RESP states in figure 3.8. After connection, the task moves to the data transfer section, comparable to the DATA_TRANSFER state in figure 3.8.

Just as the suspension on the entries matches the STATE variable of the FDT, so the acceptance of legal entries corresponds to a legal transition of states.

The ADA specification of the BUFFER_MANAGER and of the ROUTER tasks and ROUTER_MANAGER have been omitted, but their specification proceeds as per the examples given.
package body TS_ENTITY is

    task DISPATCHER is

        entry GET_MANAGER(  
            TO_T_ADDRESS : T_ADDRESS_TYPE;  
            FROM_T_ADDRESS : T_ADDRESS_TYPE;  
            QUALITY : QOTS_TYPE;  
            BUFFER CREDIT : BUFFER CREDIT_TYPE;  
            MANAGER_ASSIGNED : MANAGER_PTR;  
        ) out ( ok, disconnect ));

        entry T_CONNECT_IND(  
            TO_T_ADDRESS, FROM_T_ADDRESS; out ... ));

        entry FREE( ... );  
        -- used by managers to report when free;

        entry TPDU_CONNECT_IND( ... );  
        -- Used by mapper to indicate remote connect ind's

    end DISPATCHER;

    task body DISPATCHER is separate;

    task type CLASS_0_MANAGER is

        entry CONNECT_req( TCEP : TCEP_identifier_type;  
            to: T_ADDRESS, fr: T_ADDRESS, in T_address_type;  
            QOTS_request : in Quality_of_TS_type;  
            options : in option_type;  
            TS_connect_data : in TS_connect_data_type)

            entry CONNECT_response(...);

        entry CONNECT_confirm(...);

        entry DISCONNECT_req(...);

        entry DATA_req(...);

        entry EXPIDITE_DATA_req(...);

        entry DATA_ind(...);  
        --expidited_data_indications also

        entry TPDU_CONNECT_CONFIRM(...);

        entry TPDU_DISCONNECT_IND(...);

        entry TPDU_DATA_IND(...);

        entry TPDU_EX_DATA_IND(...);

        entry TPDU_REJECT_IND(...);

    end CLASS_0_MANAGER;


FIG 3.12(a): ADA SPECIFICATION OF TRANSPORT ENTITY - DISPATCHER AND CONNECTION MANAGER INTERFACE
task BUFFER_MANAGER is
    entry CREDIT( NUMBER_BYTES : in NUMBER_BYTES_COUNT;
               RESULT : out RESULT_TYPE);
    entry RELEASE( NUMBER_BYTES : in NUMBER_BYTES_COUNT);
    entry GET_BUFFER( ...);
end BUFFER_MANAGER;

task BUFFER_MANAGER is separate;

-- task type ROUTER_IN is
    -- Also strips TRANSPORT PDU's and determines type and manager, then
    -- calls the appropriate TPDU... entries in DISPATCHER or MANAGERS.
    -- One manager for each network connection open.
end ROUTER_IN;

task body ROUTER_IN is separate;

-- task type ROUTER_OUT is
    -- this task, one for each network connection open, builds the TPDU
    -- requested by the manager task, then passes it to the network for
    -- onward transmission. It may be accessed by various managers if
    -- multiplexing occurs.
end ROUTER_OUT;

-- task ROUTER_MANAGER is
    -- And of course, a manager to assign tasks to new network
    -- connections as they are needed.
    -- Details left out
end ROUTER_MANAGER;

procedure T_CONNECT_REQ( TCEPI: TCEPI_IDENTIFIER-TYPE; ... ) is begin
    DISPATCHER.GET-BUFFER( TCEPI, ... , result );
    if TCEPI. MANAGER /= NULL
        then TCEPI. MANAGER. CONNECT_REQUEST( ... ); end if;
    if result /= ok then reason := ... ; end if;
end T_CONNECT_REQ;

procedure T_CONNECT_IND( ... ) is begin
    DISPATCHER. CONNECT_IND( ... );
    TCEPI. MANAGER. GIVE_CONNECTION_INFO( THIS_END, OTHER_END, TCEPI, ...);
end T_CONNECT_IND;

procedure T_DATA_REQ( ... ) is begin
    if TCEPI. MANAGER /= null then TCEPI. MANAGER. DATA_REQ( ... );
    if result = DISCONNECTED then TCEPI. MANAGER := null;
    else
        result := ILLEGAL_CONNECTION;
    end if;
end T_DATA_REQ;

-- other procedures similar

FIG 3.12(b): ADA SPECIFICATION OF TRANSPORT ENTITY
- OTHER TASKS PLUS INTERFACE PROCEDURES
separate ( IS_ENTITY );
task DISPATCHER is
  use UNION FIND SET PACKAGE;
  MAX_NUMBER_OF_CONNECTIONS : integer const := 256;
  FREE_CONNECTIONS : integer set ( MAX_NUMBER_OF_CONNECTIONS );
begin
  for i in CONNECTION_TYPE LOOP INSERT( i, FREE_CONNECTIONS ); END LOOP;
  loop begin -- begin to allow local exception handling
  select
    accept GET_MANAGER( ... ) do
      -- check addresses, buffer credit, free connection
      CEPI := ( CONNECTION := FIND( FREE_CONNECTIONS );
      MANAGER := NEW ( CLASS_O_MANAGER ); end GET_MANAGER;
    or accept TPDU_CONNECT_IND( ... ) DO ... -- check address, etc
    select
      accept T_CONNECT_IND( ... ) ...;
    else
      TRANSPORT_OUT. TPDU_DISCONNECT_REQ( ... );
    end select;
    or accept FREE( i ); FREE_CONNECTIONS := i UNION FREE_CONNECTIONS;
  end select;
  exception when ... => ..., etc.
end; end DISPATCHER;

FIG 3.13: ADA SPECIFICATION OF TRANSPORT ENTITY
          DISPATCHER SKELETON

3.4 MAPPING FDT CONSTRUCTS TO ADA DESIGN CONSTRUCTS

The previous section showed how a specific FDT design

   can grow into a complete ADA specification in a direct

d manner. This section gives some rules for this mapping.

3.4.1 FDT MODULE's

The ADA specification equivalent of an FDT MODULE is

 normally a TASK or a PACKAGE. If the MODULE has a straight-
 forward structure and no concurrent activities to manage,

 then it will map directly to a task. If it has divergent
 concurrent activities, then it should be an active package.

 A module in the middle, such as will be discussed in the
 next section, will likely become a task with transport tasks
 assisting. This collection of tasks should be enveloped in
 a package, so the complete MODULE will still be an active
 package.
separate( TS_ENTITY );
task body CLASS_0_MANAGER is

type SDU_TYPE = record
  next: access SDU_TYPE;
end record;

SDU_SYMBOL: record
  SDU_TYPE;
end record;

THIS_END, OTHER_END: CONNECTION_ID;
ME: CLASS_0_MANAGER_PTR;
PDU_ROUTE: access ROUTER_OUT;

begin
  -- init things
  select -- now it's connection setup time
  accept CONNECT_REQ( ... ) do
    ROUTER_MANAGER. ASSIGN NS CONNECTION_MANAGER( ... PDU_ROUTE ... );
    ROUTER_OUT. TPDU_CONNECT_REQ( ... );
  end select;
  select accept TPDU_CONNECT_CONF( ... );
  or accept TPDU_DISCONNECT_REQ( ... );
  result := disconnect;
  or accept CONNECT_REQ( ... ); ROUTER. TPDU_DISCONNECT_REQ( ... );
  end select; end CONNECT_REQ;
  or accept GIVE_CONNECT-INFO( ... );
  select accept CONNECT_CONFIRM( ... );
  or accept DISCONNECT_REQ( ... );
  or accept TPDU_DISCONNECT-IND( ... );
  end select; -- finished connecting call;
while not disconnected loop;
  select accept DISCONNECT_REQ( ... ) do
    ROUTER_OUT. TPDU_DISCONNECT_REQ( ... );
    BUFFER. FREE( ... ); DISPATCHER. FREE( ... );
    end DISCONNECT_REQ;
  or accept TPDU_DISCONNECT-INDICATION( ... ) do ... end TPDU_DIS...
  or when SDU_INDICATIONS =? null
    accept DATA_IND( ... ) do DATA-OUT := NEXT( SDU_INDICATIONS );
    end;
  else delay( 'TIME_LEFT' ) do ... end; -- retransmit outstanding PDU's
  end select;
end loop;
exception
  when TASKING_ERROR => ..., etc.
end CLASS_0_MANAGER;

FIG 3.14: ADA SPECIFICATION OF TRANSPORT ENTITY
- CLASS 0 MANAGER SKELETON

In the simple case, a module may become a procedure, or a collection of procedures. This was the case for the interface module which became procedures CONNECT_REQ, etc. in the package TS_ENTITY, since the interface was there only to permit all entities to use a single channel.

3.4.2 FDT INTERACTIONS

The second major area of concern when mapping FDT MO-
RULES to structured design objects is to identify the target of FDT INTERACTIONS.

FIG 3.15: IMPLEMENTING FDT CHANNEL WITH INTERACTIONS

If the module is a package, then the interfaces to the package should be procedural. It is possible to place visible tasks in a package specification, but this begins to
destroy the sanitary interface concept of design. Under the hierarchical approach, the package must not call the user, so all FDT interactions of the server modules must be specifically requested in a procedure call (e.g. DATA_IND) or returned as a parameter of other calls (e.g. DISCONNECT_IND is a possible result in all other calls).

If the module maps to a task, then the interactions become entries, as shown in mapping figure 3.5 to the structure graph, figure 3.10. Figure 3.15 shows the various possible Ada inter-task communication schemes equivalent to the FDT "Channel". Consider an FDT channel between two modules T1 and T2, as shown in figure 3.15, where T1 activates interactions a & b in T2, and T2 activates c & d in T1. In the first scheme, the interactions between T1 and T2 all become entries in T1, where T2 must call the correct entry to get an interaction. In that scheme, T2 must know which interaction is next, which may be inappropriate, so a second scheme has each task calling the appropriate entries in each other to activate an interaction. This scheme can create deadlock, so the third scheme retains the owned entry structure, but places a messenger or transport task in between the tasks to collect transmissions one way from T1, and deliver them to the correct entry in T2. Of course, additional transport tasks, mailbox tasks, etc. can be substituted, as will be discussed in the next chapter.

If the module maps to a procedure, or a collection of procedures, then the interaction becomes the procedure call itself, and the interactions activated become the calls the procedure activates. In the case of the procedure, the channel that the interaction occurred on is a parameter. This is not in general true for task entries, since each in-

* This transport task should not be confused with tasks belonging to the transport entity.
interaction on each channel usually requires a dedicated entry.

3.4.3 STATE TRANSITIONS

As mentioned above, it may be the case that a module is so complex that the only suitable object for it is an active package. At that point, there is likely to be many minor states as opposed to an identifiable major state transition. Such a situation occurs in in Bochmann's second example of a TRANSPORT ENTITY MODULE [2(ANNEX 2)], shown in figure 3.15. In a case such as this, the task, or tasks, inside the package will reflect the appropriate state transitions of the module.

If the module is represented as a task, then there are two basic ways of preserving the state and transition information of the FDT specification. One way was already demonstrated in the preceding section, that of specifying the task such that the algorithm itself implicitly maps the state transitions. The other method is to maintain the concept of a state variable in the task called, say STATE. The task would then consist of one large LOOP .. END LOOP construct, with a CASE STATE IS ... construct to pick the correct state for this loop iteration. A state transition induced by an interaction then becomes the acceptance of an entry (by the task or by a called task) and the new state is set before completing the code for that entry block.

The third possible mapping of a module was a collection of procedures. Each procedure can represent one or more states. The first is the idle state, waiting for a call. The placing of an entry call is equivalent to moving to another state. These collection of procedure calls really only represent minor states, however, as procedures in ADA are by nature re-entrant. If a collection of procedures represents a module, then, that module can adjust minor states
FROM THE ISO FDT TO SYSTEM DESIGN

of its calls, but can only have one major, dummy, state.

3.4.4 SUMMARY

This completes the section detailing the mapping of ISO FDT specifications onto ADA specifications. There are other, non-isomorphic mappings which can be generated, but to examine these at this point clouds the logical design issue. The next section will examine Bochmann's development of an FDT specification of a multi-connection transport module, and possible derived designs.

3.5 USING THE ISO FDT: A THIRD EXAMPLE

3.5.1 INTRODUCTION

Section 3.2 extended Bochmann's example of co-ordinating modules implementing a single connection entity to co-ordinating modules implementing a multiple connection entity. Bochmann's own example solution to the multiple connection entity [2(ANNEX 2)] was to use one module to implement the complete entity. This entity services many tasks and provides class 0 and class 2 services to session entities. The module may be described as monolithic, since all connections are processed as integral data units within the module.

This section will examine the nature of alternate structured designs derived from the above formal specification, and discuss how design commitments are really made at the FDT level.

The monolithic design presented by Bochmann was not intended to represent a preferential way of describing a multiple connection manager. Its presence and form in a formal specification, however, can lead to serious design deficien-
3.5.2 OVERVIEW OF THE MONOLITHIC ENTITY

Figure 3.16 shows the pictorial representation of the multiple connection module specified by Bochmann. Each TSAP is a copy of the TS_ACCESS_POINT of figures 3.3 and 3.4. Each NSAP is a copy of a NETWORK_ACCESS_POINT. The TS_ENTITY module interacts with these two sets of channels, and performs all management functions internally.
To perform all actions internally, the entity must maintain an array of state variables, one for each connection. Since a state on one connection cannot be permitted to interfere with transitions on other connections (the states are independent), only one major DUMMY state is permitted. Similarly, the entity must respond to all interactions immediately, so all potential state transitions must be open at all times.

Along with the state of each connection, the module also maintains mappings for the T-CONNECTION $\leftrightarrow$ N-CONNECTION interface, and flow control info and class of service for each connection. The net result is an array of records, each describing the exact condition of a connection. As transitions occur, the module takes action and updates the proper record accordingly.

A skeleton of the structured design for Bochmann's FDT specification for a multiple connection transport entity is given in figure 3.17. A discussion of this design follows in the next section.
module TP_ENTITY( NSAP: array[ N_ADDRESS_TYPE ] of NS_PRIM( USER, SERVER ));
   TSAP: array[ T_ADDRESS_TYPE ] of ...
   send_buffer, receive_buffer: array[ TC_ID_TYPE ] of ...
end;

var TC: array[ TC_ID_TYPE ] of record
  state: ( closed, wait_cc, op_in_prog_calling, ... closing);
  local_t_address, remote_t_address: T_ADDRESS_TYPE;

  assigned_NC: NC_id_type;
end;

NC: array[ NC_id_type ] of record
  NC_state: ( closed, open_in_prog, open);
  remote_n_addr: N_ADDRESS_TYPE;
end;

state( dummy );

function determine_add_address( t_addr: ...; N_addr: ...): Additional_address_info
... ( other functions & procedures to
  - match TC id's <-> NC id's,
  - determine PDU size, length, build PDU, etc. )

(TTRANSITIONS)
from dummy to dummy (* REQUIRED BY FDT SYNTAX*)
(* GENERAL PURPOSE TRANSITIONS*)

(* RECEIVE NSAP *)
any N_addr: N_ADDRESS_TYPE do
when NSAP( N_ADDR), N_DATA-IND DO WITH NC( find_NC_id( N_ADDR, NCEP )) DO
  provided -- test flow control, class ok, buffers ready, ...
    begin
    ... decode
    with received_PDU do begin
    CASE TC[TC-id].state of
      closed: .. (determine reason for error and build disconnect PDU)
      wait_NC, op_in_prog_calling, op_in_prog_calling, open:
        case code of
          CC: if TC[tc_id].state # op_in_prog_calling then prot_error
        else begin .. connect, set up records, build & send PDU
        DR: ... (disconnect logic)

(* CONNECTION ESTABLISHMENT *)
any T_addr: T_ADDRESS_TYPE do
when TSAP( T_ADDR), T_CONNECT-REQ
  provided ... ok ... set up parameters, finding NC etc. in another part
  provided ... not ok ... TSAP[local T_addr]. DISCONNECT-IND;
    ... state := closed;

FIGURE 3.17: SKELETON OF MULTIPLE CONNECTION TRANSPORT ENTITY
(DUE TO BOCHMANN)

3.5.3 ISSUES IN MAPPING MONOLITHIC SPECIFICATION TO DESIGN

3.5.3.1 ISOMORPHIC DESIGNS -

The FDT specification of the single monolithic manager
just given is strongly reminiscent of the monolithic X.25
design of section 2.3. Figure 3.18 shows a structure graph
design of a similar transport task. There is a natural iso-
morphism between the FDT of figures 3.15 and 3.17, and the design figure 3.18, in that:

1. The connection state record of the FDT becomes a state record in the ADA design.

2. The network connection records become an array of records in the ADA design.

3. Each interaction of a connection in the FDT becomes an entry in the ADA design, or is the result of an entry (e.g. disconnect). In this vein,

 ANY T-ADDR DO WHEN TSAP(T-ADDR) DO ...

 becomes

 WHEN TSAP-REC(T-ADDR).OPEN => ACCEPT TSAP[ .. ]

 ...

4. Interactions with the network connections are handled by transport tasks which pass the PDU to a single entry N-DATA-IND. This is the only activating interaction from the network recognized by the FDT module.
The verification of this design in light of the FDT specification is fairly straightforward. Each FDT component maps straightforwardly to a counterpart in the ADA specification. The transition and resulting actions taken by the FDT module after an interaction are essentially atomic. They are reflected by indivisible task manipulations in the ADA design.
task TP_ENTITY is
  entry CONNECT_REQ (to, from: ..., CEP_ID: out CEP_id_type);
  entry CONNECT_CONF (CEP_id) (to, from: ...); -- family
  entry CONNECT_RESP (to, from: ...);
  entry DATA_IND (CEP_id) (...); -- family, 1 for ea TCEP
  
end TP_ENTITY;

begin

  type ... (definitions for transport protocol exchange)
  type TC_type is record
    state: (closed, wait_cc, op_in_prog_calling, ... closing);
    local_t_address, remote_t_address: T_address_type;
  end;

  tc: array (TCEP_id) of TC_TYPE;
  type NC_type is record
    NC_state: (closed, open_in_prog, open);
    remote_N_address: N_address_type;
  end;

  nc: array (NC_id) of NC_type;

end;

function determine_addr( T_addr: ...; N_addr: ...): Additional_addr_info;

... (other functions & procedures to
  - match TC id's => NC id's,
  - determine PDU size, length, build PDU, etc.)

BEGIN
  LOOP

    accept N_DATA_IND (...)
      do
        determine Id of TC record
        -- test flow control, class ok, buffers ready, ... begin
        ... decode
        CASE TCI[T-CID].state of
          closed: ... (determine reason for error and build disconnect PDU)
        wait_NC, op_in_prog_calling, op_in_prog_called, open:
        case code of
          CC: if TCI[T-CID].state & op_in_prog_calling then prot_error
        else begin ... connect, set up records, build & send PDU
        DR: ... (disconnect logic)

      end

      or when TSI[ADDR].state=open =>
      accept T_CONNECT_REQ (T_ADDR) (...);
      do
        provided ... or ... set up parameters, finding NC, etc. in another p =>
        provided ... not ok, put disconnect PDU to proper NC record
      ... state := closed;

  end

  ...}

FIG 3.19: ISOMORPHIC MONOLITHIC MANAGER
- SKELETON ADA DESIGN SPECIFICATION
3.5.3.2 ATTEMPT AT A NON-ISOMORPHIC DESIGN

Should a design team wish to implement a multiple connection transport manager on a distributed system, or if they had at their disposal a multi-tasking operating system such as Intel's RMX[11], then the monolithic design of the previous section is not suitable. They would have to take the FDT, and create a distributed design to meet its specifications.

Figure 3.20 reflects essentially a non-isomorphic design of the FDT module of 3.4.1. It is strongly reminiscent of the design of 3.2. In this design, each connection record of the FDT becomes a task. Its major state is the state of the connection. Similarly, each network connection descriptor record becomes a task.

The assignment of connection manager tasks to a connection was an implicit function in the FDT module. Here it must be managed by a dispatcher.

The interfacing between the various tasks in this ADA design is next. Here the FDT is of little assistance. There are procedures and functions to keep straight the appropriate interactions between transport connections and network connections, and to manipulate data between them. In the FDT module, these were essentially atomic operations: in the current design, these must be handled by entry calls and ACCEPT's, which are non-atomic and, unless the designer is very careful, could result in strange anomalies not in the original design, such as: if a network connection task calls an entry that isn't accepted, which now is responsible for generating the disconnect PDU?
FIG 3.20: NON-ISOMORPHIC DESIGN FROM MONOLITHIC MANAGER

The major difficulty here is that the FDT system is specified in a quasi-sequential manner, while the second design attempt is highly concurrent. It is a non-trivial process to convert sequential algorithms into concurrent ones, a process which has so-far eluded some of the best minds in
the world, hence one should not expect to readily convert such a monolithic into an equivalent concurrent one. It promises to be a major undertaking for any design team.

Once a design is created, it must be verified against the original specifications. Here again, the non-isomorphism of the second design creates problems. The tasks have replaced records and imbedded code of the module. It is not sufficient to ensure that the codes do the same thing: a verification must also show that other dependencies or errors have not been induced. These could come from time, relative order of events, competition for a single entry or task (such as may come from multiplexing over a network connection, etc.). The complete data orientation and atomic nature of the FDT under consideration makes this verification very difficult.

It is clear then, that the monolithic module specification just examined in this section does strongly favour designs which are naturally isomorphic to it, and discourages designs which are not. It is for these reasons that the FDT as proposed can be considered to be really the first stage of the design stage, similar to the DATA FLOW stage of chapter 2.

3.6 Conclusions

Although attempting to specify a formal description of OSI systems without influencing design or implementation, it is evident that the Formal Description technique, as proposed, is too close to the design process for it not to have a profound effect on design. Unless the description is chosen carefully, decisions made at the Formal Description level as to module structure, choice of interactions, etc. lead to commitments that are difficult to deviate from at the design and implementation level. This serves as an ex-
cellent example of the ease with which designers can fall into traps in the design process.
CHAPTER 4
FURTHER ISSUES IN OSI DESIGN

4.1 INTRODUCTION

The previous 2 chapters showed how to design Network and transport layers using structured design techniques and the Ada programming language from both data flow analysis and OSI Formal Description Techniques. The design approach in all cases up to present was hierarchical, with users calling the services of the next layer for all interactions, requests or indications.

This chapter examines alternate approaches to the design of such systems, including non-hierarchical designs, reflects further on the formal description and design process, and discusses the use of Ada as a specification language.

4.2 NON-HIERARCHICAL CONTROL DESIGNS

4.2.1 OSI FDT ISOMORPHIC DESIGN

4.2.1.1 Mapping Interactions to Tasks -

The proposed OSI FDT uses the concept of channels between two entities, which may contain zero to an infinite number of pending service units between modules in either direction. If both entities are tasks, then emulating the channel using Ada constructs is fairly straightforward, as discussed in the previous chapter, and in figure 4.1. A dedicated mailbox task with an internal queue for each di-
rection and an in and out entry for each task will suffice often. If however the communicating tasks also have their own entries that they must service, they cannot afford to suspend on such a mailbox task. In addition, as shown in chapters 2 and 3, the design may have many tasks passing messages over the same SAP, since these designs had one manager and a ROUTER_IN task for each connection.

Splitting the mailbox task into two messenger tasks, each calling the appropriate entry in the correct task is conceivable, but suspension on one of those entries has a domino effect, and still causes undesired waiting on behalf of the caller. In addition, such a structure limits the queuing available, since this task must deliver each service data unit immediately as it does not know when the receiver will be ready for the data unit.

4.2.1.2 A Message Oriented Design -

The most general solution to the above mentioned difficulties is to maintain the mailbox structure of figure 4.1(a), and the multiple entries of each communicating task, one entry corresponding to each interaction specified by the OSI FDT as activating that module. A messenger task is now required to wait for any such messages on the appropriate MAILBOX_TASK out entry, and to deliver that service data unit to the correct entry in the proper task. Figure 4.1(c) shows the general method just described.
Figure 4.2 shows many tasks in two entities communicating via a SAP task. The SAP task has many entries, one for each connection input in each direction. A messenger task waits on the appropriate entry for a service data unit from the mailbox task, then calls the appropriate entry in the destination task corresponding to the FDT interaction.
Calls to the appropriate "in" entry are made by the calling task itself, or another MESSENGER_OUT task may be used.

**FIG 4.2 IMPLEMENTING A SAP TASK WITH ADA CONSTRUCTS**

In this design, the mailbox task is a visible task in the interface of the layer package. Figure 4.3 gives a skeleton outline of the interface specification of such a layer package. Note that the lower entries to the mailbox task, used by entities in the package, are visible to the entity outside the task, hence the possibility of erroneously calling the wrong entry exists. A possible solution to this state of affairs is to include an extra parameter in the calls to the entries used by the internal entity. This parameter would be of type "limited private", hence inaccessible to outside users and hence could be used to protect these entries. No similar mechanism exists in Ada to protect the upper entries when this SAP task resides in the layer package.
package LAYER_N is
  task type SAP_TYPE is
    entry USER_IN(connection_type)
      (msg: .......) if
    entry USER_OUT( CONNECTION_TYPE )
      (msg: .......) if
    entry SERVER_IN( CONNECTION_TYPE )
      (msg: .......) if
    entry SERVER_OUT( CONNECTION_TYPE )
      (msg: .......) if
    end SAP_TYPE;
  N_SAPS: ARRAY( number_saps ) of SAP_TYPE;
  end LAYER_N;

FIG 4.3: INTERFACE SPECIFICATION OF A SAP TASK.

Another possible design is to remove the SAP task from the layer package completely and have a general interface consisting of SAP task types and appropriate limited private data types. When a task is required for inter-entity communication, it can be created and the access variable pointing to it given to both users, along with an appropriate value of the data type, different for each user. Now no other entity can access the SAP at all, since its access variable is available only to the proper entities, and the communicating entities cannot access the wrong entries, since each group of entries for one entity, user or server, can be protected by a limited private data type. Figure 4.4 gives the interface specification for such a package. The server would call GET_SAP_FOR_SERVER which would return a SAP and the user's protection variable. The user through management contact with the server entity, would get a pointer to the SAP from the server, and call GET_USER_PROTECT to get his protection variable for use of that SAP.

Mechanisms such as the ones just proposed allow the designer to map the proposed OSI FDT isomorphically into Ada
design specifications. These designs are not hierarchical in nature; rather they are closer to a Data Management design, or a message passing design. The SAP's just mentioned could reside inside the server's layer package, reside between layers, or even be assigned out of a pool of SAP's held by the SAP_PROVIDER package. A distributed design would tend toward the first mechanism, while a non-distributed design would tend toward the second or third

```plaintext
package SAP_PROVIDER is
  type SERVER_ENTRY_LOCK is limited private;
  type USER_ENTRY_LOCK is limited private;
  task type SAP_TYPE is
    entry USER_IN(connection_type)
      (msg: ... lock: USER_ENTRY_LOCK ... );
    entry USER_OUT(connection_type)
      (msg: ... lock: USER_ENTRY_LOCK ... );
    entry SERVER_IN(connection_type)
      (msg: ... lock: SERVER_ENTRY_LOCK ... );
    entry SERVER_OUT(connection_type)
      (msg: ... lock: SERVER_ENTRY_LOCK ... );
  end SAP_TYPE;

  N_SAPS: ARRAY(number_saps) of SAP_TYPE;
  procedure GET_SAP_FOR_SERVER( ... 
    mailbox: out access SAP_TYPE;
    my_key: out SERVER_ENTRY_LOCK);
  procedure GET_KEY_FOR_USER( ... 
    mailbox: in access SAP_TYPE;
    user_key: out USER_ENTRY_LOCK);

  private
    USER_ENTRY_LOCK, SERVER_ENTRY_LOCK : integer;
end LAYER_N;
```

FIG 4.4: EXAMPLE OF A SAP PROVIDER PACKAGE

4.2.1.3 A Piatkowski-derived design -

In his proposals for the formal specification of OSI nodes, Piatkowski[13] proposed a system of Ports, Routers, and communicating Managers, where managers negotiate all connections and ports handle all data units exchanged. A structure diagram interpretation of such a system is given
in figure 4.5. In this system, each port manages several connections, although Piatkowski's statement that there should be one port for every node reachable in the system makes it very ambiguous as to exactly what each port represents. If one port exists for each node reachable, then there is no correspondence between PORT's and SAP's nor between PORT's and CONNECTIONS. It seems more plausible to make a port correspond to a SAP, which makes the proposed design more like the proposed OSI FDT.

**Fig 4.5: A Piatkowski inspired OSI Layer Design**

In a design such as this, the manager plays a much larger role in the peer-level negotiations than in designs.
presented in previous chapters and in previous parts of this chapter. Here connection negotiations are exchanged between managers. This issue will be discussed in more depth in the next section.

The design just presented has been another example of a Data Management design structure as opposed to hierarchical designs which are the thrust of this document. No comment nor comparison is intended as to the relative merit of any of the designs or design techniques.

4.3 CONNECTION MANAGEMENT ISSUES

4.3.1 GENERAL

In his proposal, as mentioned above, Piatkowski proposes a management function which performs all connection management negotiations. This approach does solve some difficulties arising out of virtual circuit manager designs presented earlier, but does create difficulties in other areas.

4.3.2 DISPATCHER - VIRTUAL CIRCUIT MANAGER DESIGN

In the previous chapter, the distributed TRANSPORT ENTITY example had each virtual circuit manager, once assigned, negotiate all aspects of the connection. In the case where only one class of service is provided by the entity this will suffice; however, when a requestor may specify a desired class of service, but through end-to-end negotiations another class is chosen, this design is not sufficient. In order to work now each virtual circuit manager must be capable of handling any class of service requested, or be able to hand off the connection to another class of manager. The first method is not very satisfactory since it is more natural and more modular to create each class ser-
vice as a different task. The second example obfuscates the clean break between the dispatcher and the managers, since the dispatcher would now have to be involved in re-assigning managers and notifying the user, or the Virtual Circuit Managers would have to do some of the updating of manager-connection tables, which should be the pervue of the dispatcher.

4.3.3 MANAGER - PORT DESIGN

A design similar to Piatkowski's would have the manager handle all negotiations on the connection. It would therefore finalize negotiations on the making of a connection, then pass the data managers identifier (access variable or array index) to the user and router for data processing. Any resets, changes, or disconnecting of the connection are also the pervue of the Connection Manager.

As intimated earlier, there are also difficulties with this design approach just presented:

1. When a disconnect indication arrives, the manager must notify the user. Unfortunately, the user will be expecting data from a port task, hence would be unavailable for such a notification. Other schemes, such as having the user time-out and check for disconnect, passing the disconnect indication to the port, etc. are all possible, but not sanitary.

2. A manager should be in overall control of the entity or layer, assigning SAP's or connection managers, collecting statistics, recovering from severe errors, etc. It should not be involved in routine negotiations with peer entities, since the chance of it becoming involved in a severe error increases dramatically, leaving the complete entity locked up or unusable.
3. Such a manager approach seems to imply a much larger reliance on common memory systems, at least at the entity level, since by nature control is much more tenuous in highly distributed systems. A unitary manager attempting to control all connections in a distributed system would like-wise be tenuous. In the Dispatcher/VCM design, the relationship between the dispatcher and the managers is infrequent and tenuous, hence more suited to distribution.

4.3.4 DISCUSSION

From the above discussions, it is obvious that either method investigated has its advantages and disadvantages in any moderately complex system. Evaluating the relative merits of each method and selecting one for the design is clearly one of the major decisions that a design team would be faced with. Given certain circumstances, hardware, software, development and operating systems, one method or other would likely be easier or "better". It is inappropriate at this time to recommend one or the other, only to note that the trade-offs exist and are clearly stated.

4.4 THE USE OF CHANNELS FOR INTER-MODULE COMMUNICATION

In Bochmann's paper to the ISO, the FDT technique used proposes that all interactions between modules occur over channels, with a channel defining a one-to-one relationship between modules. These interactions are to define the interactions of a module with its environment.

The interaction described above is good at emulating a Service Access Point, since all SAP's define a one-to-one relationship between entities. They are not good at describing what occurs inside an entity, since a module may provide a variety of services to different other modules.
For example, the dispatcher must manage many Virtual Circuit Managers, which communicate with the manager only rarely to notify it when they are free. To specify a formal channel of interactions between such modules as the dispatcher and each virtual circuit manager is not appropriate, as they greatly increase the perceived complexity of the system without deriving any major benefit from their usage. It also makes any diagrams or formal descriptions of such systems more complex and confusing, hence serves to cloud rather than clarify the formal specification.

A better approach for the OSI FDT would seem to be to use channels for the interactions between entities, such as SAP’s provide, but to use open Ada-like rendezvous connections for the internal (sub-entity) communications. This seems less cumbersome than to stick rigidly to the channel concept throughout.

4.5 SUMMARY

This chapter has presented some issues in formal specification, design and implementation. As more experience is garnered in the use of such designs, the relative merit of some of these concepts will be born out. The next chapter will examine implementation issues in more detail while presenting some possible implementations of the hierarchical designs of previous chapters.
CHAPTER 5
IMPLEMENTING AN OSI DESIGN

5.1 INTRODUCTION

Up to this point, this document has been considering design in an abstract way, without much concern for implementation issues. This chapter closes the design loop by considering how the major OSI designs may be implemented on existing systems available at Carleton University and presenting some general ideas for such implementations. No actual implementations were built by the author up to this point, but the methodologies presented are straightforward and there is a potential to convert the skeleton designs proposed herein to actual, working designs.

5.2 GENERAL DIRECTIONS IN IMPLEMENTATION

5.2.1 POTENTIAL IMPLEMENTATION SCHEMES

The recommended direction in design of this thesis has been toward hierarchical, highly distributed designs. The actual implementation could be hierarchical or non-hierarchical, distributed or monolithic. A lot of the choice in choosing a particular implementation will be governed by the design and support systems available, the implementation support systems available, the type of system desired (distributed or co-resident), and the degree of potential future enhancement forseen.

Piatkowski points out[13] that, as OSI concepts become
more clearly defined and accepted in the systems community, vendors will appear with one particular layer or entity available on a board. OSI system implementers cognizant of this fact will likely strive to logically separate the various entities as much possible to allow future modifications to their systems to include migration of parts of the design to such "boards". For this reason, all discussions and suggested implementations in this chapter will be highly modular at the entity level.

5.2.2 The Hierarchical System

Figure 5.1 shows a structured diagram of a highly distributed system. Each entity of the OSI node is resident on its own hardware, complete with its own dedicated CPU, I/O ports, operating system, etc. They may share common busses with the host system and/or each other, or may alternately be connected by "thin wire" connections. The key features of this implementation is that each entity has its own resident operating system kernel, and the interface to the Host System is restricted to physical ports and device drivers. The result is that the choice of hardware, resident operating system, and development system for each "board" is independent from the host environment.

Operating systems for such dedicated hardware entities should be of a stand alone nature, and multi-tasking at a low level. Two existing systems available at Carleton University fall into this classification: iRMX 80 and 86 for the 8080, 8085 and 8086 microprocessor chips, and Multi-pascal for the PDP-11 and LSI-11. Both of these systems, while not for sharing a CPU with a more standard Oper-

"Multi-tasking at a low level" means that tasks will resemble ADA tasks, not programs or entities as happens in many current multi-tasking systems, such as RSX, C, MPM, etc.
ating system such as CPM or RT-11, do provide low level multi-tasking and as such, permit the implementer to map the distributed designs nearly isomorphically onto the implementation.

Figure 5.1: HIGHLY DISTRIBUTED OSI NODE

5.2.3 Co-resident Implementations

Figure 5.2 shows an OSI implementation resident on the host system and sharing system resources with users. The key feature about this style of implementation is that no new hardware is required, and that the hosts resources can be used to develop and test the system. The implementation shown in Figure 5.2 features each entity as a separate task.
in a multi-tasking environment. This is done to reduce the complexity of the overall OSI implementation, and to permit future upgrades by simply removing a software entity and substituting a dedicated hardware system. More will be said about this approach later.

**FIGURE 3.2: OSI NODE RESIDENT ON EXISTING SYSTEM**

In the OSI implementation of figure 3.2, each entity must be friendly with the underlying, monolithic host operating system. It cannot be permitted to access terminals, clocks, or external lines directly, but must use standard operating systems and calls to perform these tasks. For
this reason executives such as iRmx 80 and Multi-Pascal cannot be used to generate entities isomorphic to the distributed entity designs presented in chapter 3. Since there are no suitable executives such as these which are friendly to the operating system that are currently available, such implementations turn out to be monolithic software supported by handlers and interrupt service or completion routines.

5.3 IMPLEMENTING "STAND ALONE" OSI SYSTEMS

5.3.1 Using iRmx 80

5.3.1.1 Emulating ADA entries*

Intel's iRmx 80 is a multi-tasking executive for Intel 8080 and 8085 micro-computers which uses messages and arbitrary mailboxes called exchanges for inter-task communications. Messages are sent to exchanges where other tasks await for messages. A task may create, own or use any number of exchanges, but can await a message on 1 at a time.

An entry call corresponds to a SEND-WAIT, RECEIVE, REPLY message construct. Figure 5.3 shows how an ADA entry maps onto an iRmx 80 system. Two exchanges are needed to provide the most basic entry construct. The calling task performs an

ROSEND( T.E, msg )

of a message to the exchange corresponding to the entry, then performs an

REPLY$MESSAGE=REQWAIT( REPLY$EXCHANGE )

at his own REPLY exchange. The acceptor REQWAIT's at the ex-

This section parallels R. Sharmans thesis[15] developing an Ada-like rendezvous mechanism for iRmx86, but differs in detail and in presentation.
change for calls, processes it, then terminates the rendezvous by sending the return parameters in a message to the callers reply exchange. Using this method, the acceptor can have one exchange corresponding to each entry it needs. The request messages queued on each exchange correspond to the tasks queued on an entry.

![Diagram of exchange for release of rendezvous]

**FIGURE 5.3: iRMX 80 IMPLEMENTATION OF A RENDEZVOUS**

Selective accepts are more cumbersome. Figure 5.4 shows the structure diagram of selective accepts with a time-out implemented using iRMX 80 exchanges. Since an acceptor can wait on only one exchange, it must wait on its own, private exchange, keeping track of the exchanges (entries) currently open. Tasks calling an entry must send a message to the appropriate exchange, plus a message to the task's dedicated exchange, informing it of the existence of a call, and the ID of the entry. If the exchange is open (the entry is open under "select ... accept ..."), the acceptor performs a ROACPT on the exchange, and proceeds as before. If the exchange is closed, it notes the request in a private data structure, here called ENTRIESPENDING, for future reference. Before waiting on its private exchange,
the acceptor now checks ENTRIES$PENDING for now open ones.

NOTE
Because this construction is required for the "SELECT...ACCEPT" construction, and the caller has no information as to how an entry is to be accepted this time, the acceptor must always wait at "ACCEPT" and the caller must always send two messages, even for the simple entry case discussed previously.

Figure 5.5 gives PL/M code fragments for the construction of these exchanges, plus the code to make it isomorphic to an ADA entry.

---

**FIGURE 5.4:** IRMIX 80 EXCHANGE IMPLEMENTATION OF A RENDEZVOUS - SUPPORTING SELECTIVE ACCEPTS AND TIMEOUTS
The additional entry in figures 5.4 and 5.5 are to permit the caller to abort from a call before the rendezvous is accepted. To do this, the caller must know separately when the entry is accepted, and when it is released. An ROWAIT(ACCEPTED, TIME) on the ACCEPTED entry will time the caller out if the appropriate message accepting the entry is not received. Removal of the calling message is the responsibility of the caller, and has not been included in 5.5. The acceptor now must send two messages to the caller: one to the T.ACCEPTED exchange upon commencement of the rendezvous and one to T.REPLY, containing return parameters, upon termination of the rendezvous.

DECLARE EXCHANGE*DESCRIPTOR LITERALLY 'STRUCTURE(
  MESSAGE*HEAD ADDRESS;
  MESSAGE*TAIL ADDRESS;
  TASK*HEAD ADDRESS;
  TASK*TAIL ADDRESS;
  EXCHANGE*UNIT ADDRESS);

DECLARE TASK*ENTRIES LITERALLY 'STRUCTURE(
  DEDICATED*ENTRY ADDRESS;
  ACCEPTED*ENTRY ADDRESS;
  REPLY*ENTRY ADDRESS;
  ENTRY*A ADDRESS;
  ENTRY*m ADDRESS);

CALC*ENTRY: PROCEDURE( ...) DO
  ROSEND( B.C msg );
  STAT = ROWAIT( ME.ACCEPTED, TIMEOUT );
  IF STAT.OK THEN REPLY*MSG = ROWAIT( ME.REPLY )
  ELSE ... END;

ACCEPT*ENTRY: PROCEDURE( ... ) DO
  IF SET*OPEN*ENTRIES # NULL THEN 
    ENTRY = GET*ENTRY*CALL( SET*OPEN*ENTRIES );
  IF ENTRY # NULL THEN
    ENTRY = ROWAIT( ME.DEDICATED*ENTRY );
  I F OPEN( ENTRY ) THEN DO
    PARAMETERS = ROACPT( ENTRY );
    SENDER = PARAMETERS.SENDER;
    ROSEND( SENDER, ACCEPTED );
  END;
  END;

RELEASE*ENTRY: PROCEDURE( ...) DO
  ROSEND( CALLER.REPLY, RETURN*PARAMETERS );
END;

FIGURE 5.5: PL/M CODE FRAGMENTS OF RMX 80 IMPLEMENTING AN ADA ENTRY
Each task will now be known by its entry table. New tasks can now be used reentrantly, by using the same code, and binding new entry tables and data stacks to it. This gives the implementer a virtual isomorphism between IRMX 80 tasks and ADA tasks.

Packages have no comparable construct in PL/M. To obtain the same effect in IRMX 80, an implementer must use naming conventions and comments. An example of this is, to identify all tasks in a transport entity, he could call them TS\$DISPATCHER, TS\$ROUTER, TS\$CLASS\$0\$MANAGER, etc., and to identify the appropriate procedure calls, he could call them TS\$CONNECT\$REQ, etc.

5.3.1.2 OSI Implemented Under IRMX 80

At this point, it is possible to present a possible implementation of an OSI layer N entity using IRMX 80 on dedicated hardware. Figure 5.6 gives the structure diagram of this design. Requests from entities on the N+1 layer arrive over the physical lines shown at the top of the diagram. Interrupt service routines assemble the messages and pass them to the appropriate interface task for further processing. Interface tasks act in place of the package interface procedures in the design. There must be at least 1 for each connection in use, plus one for each simultaneous request on the dispatcher. The interface tasks in turn place calls on the dispatcher or the appropriate Virtual Circuit Manager task, one for each connection in use. The rest of the implementation is straightforward, with the ADA code being replaced with the equivalent PL/M code, and the ROUTER tasks being interrupt service routines monitoring connections to

Since the task acts on behalf of the user, if a half-duplex connection exists, one per connection will suffice. In more complex organizations, up to one per procedure call per connection may be required.
the N-1 entity "board".

\[ \text{Diagram showing the design of an OSI entity, including interrupt service tasks, interface tasks, and connection manager.} \]

\[ \text{Figure 5.6: RMX 80 implementation of a stand-alone entity.} \]

The design just presented is directly isomorphic to the original ADA design, where the procedural interfaces to the packages have been replaced by hardware connections and interface tasks to fill the intended function on behalf of the requesting task. Of course, an implementer may put many such entities, N, N+1, etc., on one board by removing the appropriate interface tasks, interrupt service routines, and
IMPLEMENTING AN OSI DESIGN

substituting procedural interfaces again. No diagrams of this case are shown for iRMX 80, rather this shall be postponed to the Multi-Pascal discussion.

5.3.2 THE MULTI-PASCAL IMPLEMENTATION

Multi-Pascal[8] is a low-level multi-tasking executive suitable for implementing multi-tasking systems on stand-alone, PDP-11 or LSI-11 systems. It permits inter-process communication via user defined data structures. Multi-Pascal allows these structures to be protected from the vagaries of concurrency by providing Gladiator Monitor calls to provide sequential access to the structures by concurrent processes.

As in the iRMX 80 example just shown, if an equivalent structure to an ADA entry can be provided, then the implementation becomes just a matter of an isomorphic mapping from the ADA design to the Pascal implementation. This implementation will follow that route by showing how to map the ADA rendezvous system into Multi-Pascal monitor structures.

5.3.2.1 The ADA rendezvous in Multi-Pascal -

Figure 5.7 shows the mapping of an ADA task with two entries, A & B, into a Multi-Pascal monitor with condition

*The Gladiator Monitor, suggested by Cavers and Brown [7] implements monitor calls ENTER(gate) and LEAVE(gate) to control access to a protected region for inter-task communication, plus calls SLEEP(condition-variable) and AWaken(condition-variable) to control suspension and resumption, based on appropriate conditions.
variables A & B, and a proposed notation for this process - monitor construction. The monitor under consideration has the two condition variables A & B, mentioned before, plus one called ACCEPT for the process accepting the rendezvous to suspend on, and one called ACCEPTED for the caller to suspend on after acceptance but before release. Figure 5.8 gives fragments of code for the monitor to make the Multi-Pascal monitor equivalent to an ADA rendezvous system.

#This permits time-outs, etc to be handled effectively. The construction and rationale are similar to the IRMx 80 case which required an extra exchange for this purpose.
IMPLEMENTING AN OSI DESIGN

FIG 5.7: MAPPING ADA ENTRIES INTO MULTI-PASCAL MONITORS

NOTATION:

\[\text{Condition Variable} \quad \rightarrow\]

Parameter Queue
As shown in figures 5.7 & 5.8, a task or process wishing to place an entry call on task T, entry A, would make a procedure call

CALLxENTRY( T, T.A, parameters, timeout).

T is the record containing the monitor gate, condition variables corresponding to the ADA entries, a condition variable for the task to perform an ACCEPT, and a condition variable for the accepted task to suspend on. Procedure

ACCEPTxENTRY(T, T.openentryset, inparameters)
provides the code for the acceptor to accept an entry, while

RELEASExENTRY(T, outparameters)
provides the release mechanism. ACCEPT_ENTRY performs an ENTER( gate ), deposits the input parameters in a queue attached to the condition variable A, awakens the acceptor if it is suspended and the entry is open, then sleeps on the condition variable. When the acceptor accepts the entry, it awakens the task suspended there-on, which immediately re-suspends on the ACCEPTED condition variable. The release procedure releases the task by depositing return parameters on the queue attached to ACCEPT condition variable, then awakening the suspended task, which picks up the parameters, and returns.

NOTE
This method only permits one task to be accepted at a time. Any attempt to accept two tasks (nested accepts) would result in the tasks being suspended in a queue on ACCEPTED, while the ADA specification requires reverse release, implying a stack structure for ACCEPTED, or the release mechanism to be imbedded in the calling task's own monitor, which the acceptor will call when releasing the caller.
IMPLEMENTING AN OSI DESIGN

TYPE EntryRec = RECORD
  open : BOOLEAN;
  waiting : ConditionVariable;
  data : DataQueue;
END;

TaskInterface = RECORD
  gate : semaphore;
  openEntry : SET OF entryTypes;
  accepted, accepting : ConditionVariable;
  Entry : Array[entryType] OF entryRec;
END;

VAR Adata, Bdata, ... : TaskInterface;

{ To call an entry A in T }
WITH TData DO BEGIN
  enter ( gate );
  IF accepting. Waiting. number > 0 AND ( entry[A] IN openEntry )
  THEN awaken( accepted );
  deposit( entry[A].Data );
  openEntry := openEntry + A;
  sleep( entry[A].Waiting );
  {-- wait to be accepted }
  sleep( accepted );
  {-- now wait for end of rendezvous }
  remove( accepted. data );
  {-- get return parameters }
  leave( gate );

{ To call an entry E in T }
WITH Tdata DO BEGIN
  openEntry := . . . ;
  {-- open entries in SELECT .. ACCEPT, etc. }
  E := getNextEntry( openEntry );
  {-- select an entry from those open }
  WHILE entry[E]. Waiting. number = 0 OR ( e = NONE ) DO
  e := getNextEntry( 'openEntry ');
  IF e # NONE then awaken( entry[E].waiting )
  ELSE sleep( accepting );
  IF e = NONE then E := getNextEntry( openEntry );
  openEntry := [];
  {-- close all entries until ACCEPT }
  remove( entry[E]. data );
  leave( gate );

{ To terminate rendezvous }
WITH TData DO BEGIN
  enter( gate );
  deposit( accepted. data );
  awaken( accepted. waiting );
  leave( gate );

FIGURE 5.8: SKELETON CODE FOR MULTI-PASCAL IMPLEMENTATION OF AN ADA RENDEZVOUS

5.3.2.2 Multi-Pascal Implementation of OSI Design

The Multi-Pascal design just presented now allows the implementer to construct a system isomorphic to the ADA design specification. Figure 5.9 shows the structured diagram of such a system.

The system shown in figure 5.9 is a complete OSI node, with all layers and entities co-resident on a single Multi-Pascal system. The interface between the host and this node is via a hardware port. Interrupt service routines and interface tasks collect, decode, and call the ap-
appropriate "entry" in the proper processes monitor, just as was done previously in the iRMX 8D system. From here on, the Multi-Pascal design is isomorphic to the ADA design specification, with each ADA task becoming a process, and each entry call becoming a condition variable in a dedicated monitor. As was mentioned in the iRMX discussion, there is no structural equivalent to packages, but appropriate naming conventions for "packetized structures" and for the "interface procedures" fill the requirements. In this case, the transport interface procedures would become tsConnectReq(...), tsDataInd(...), etc.

**Figure 5.9:** SKELETON STRUCTURE DIAGRAM OF A MULTI-PASCAL IMPLEMENTATION OF AN OSI NODE
As in the iRMX 80 discussion, we may wish to split the entities among many hardware support systems. Modularity dictates that we split the system along ADA package lines, with each layer or entity migrating to a dedicated hardware "board". It may be that traffic volume exceeds the capacity of one processor for an entity. At that point an implementer is likely to add multiple processors accessing common memory to implement the packaged entity, so a discussion at the entity level, as was done in the iRMX 80 case, is sufficiently general. Here again, the iRMX 80 example already discussed shows a solid, isomorphic implementation, with the iRMX 80 tasks replaced by Multi-Pascal processes. The interface procedures become multiple dedicated tasks in the package with appropriate interrupt service routines supporting them. The remainder of the system remains unchanged.

5.4 IMPLEMENTING CO-OPERATIVE RESIDENT OSI NODES

Of the more general, user-friendly operating systems available at Carleton, none have the low level Multi-tasking requirements needed to implement an OSI node isomorphic to the ADA specification. As discussed before, the two aforementioned systems are not sufficiently compatible to these operating systems to implement a resident OSI node on a host with such an operating system resident.

Any node implemented resident on a host must be friendly to the resident operating system. As Piatkowski points out, it is also likely that future enhancements may move portions of the system to dedicated hardware. The two major constraints in implementing such a system are therefore:

1. It must be modular at the entity level: that is, it should be possible to remove a "software" entity and plug in a "hardware" entity, without having to rewrite and regenerate the rest of the entities; and,
2. It must make legal system calls for all scheduling, to interface with the external environment, for timing requirements, etc.

Since most high level languages and assemblers have ways of accessing the outside world, i.e. READ and WRITE, or equivalent, it is natural to use these mechanisms in older languages to interface one entity to others. In addition, most read and writes go to device drivers or handlers for service, and external, hardware, entities require handler(s) to interface them to the operating system. The optimum implementation is therefore to map each entity into an executable module, and to connect the software modules by pseudo device handlers.* If in the future, an entity migrates to an external board, then modification of the system amounts to removing the executable module, replacing the pseudo-handler with a proper handler, and plugging in the hardware. This is the technique used in both sample implementations, both on PDP-11 computers, one on RT-11, the other on RSX-11.

5.4.1 RT-11 Implementation

Although RT-11 single job and Foreground/Background monitors are fairly limited in scope, the Extended Memory monitor is capable of running six (6) virtual jobs, each with completion routines, etc., in addition to the fore-

*At a seminar given after this design was finalized, Intel representatives discussed their new Network and Transport implementations for iRMX 80 and 86. The Intel implementation can reside in either the local host processor environment, or on the ethernet interface board. Their recommendation was to write a suitable device handler to interface the user software with the appropriate OSI entity.
ground and background jobs standard. In the following two examples using RT-11, each layer or entity (if only one entity in a layer) will be implemented as one virtual job. Since each entity handles many events concurrently, and has no way of predicting which will be the next demand on its services, it cannot afford to be suspended on the operating system, waiting for a READ or WRITE to complete. To handle these asynchronous events, completion routines are required to notify the main routine that the operation has completed, and to set the appropriate status variable in the job.

The first implementation uses completion routines extensively to manage all activities on a connection, including making the appropriate I/O requests with proper completion routine designations. The second implementation uses the mainline as a monolithic manager to perform the protocol management necessary in an arbitrary N-entity. The completion routines here serve only to interface the manager to its environment, to set flags and queue messages, etc. for the mainline.

5.4.1.1 RT-11 Implementation - Distributed OSI Entity -

Figure 5.10 gives the structure graph of an implementation of an OSI entity using completion routines extensively. Each completion routine represents a Virtual Circuit Manager, one half responding to requests from above, the other half from the entity below. The device handler acts as the router in this implementation.

When a request is posted on a channel, the device

*The limit of six virtual jobs means that one cannot assign more than one job to a layer, hence if a layer contains more than one entity, they must be linked into one job. The diagrams and discussions assume one entity per job to simplify discussion.*
handler (the one above, or below, as the case may be) and operating system schedule the appropriate completion routine in the job to handle the request, as was directed when the .READC was issued. When the completion routine commences, it will process the request as far as is possible in a continuous fashion, as specified in the ADA language description. If a request needs to be passed on, or processed further at a later time, it will set the appropriate flags and make the appropriate monitor requests (.WRITEC, etc.), then exit. Another completion routine, or a later pass of this routine will continue the processing.

The dispatcher completion routine will monitor channel 0, where all connect requests and indications are routed by the handlers. Once it has set up the appropriate data regions for the new VCM, it will perform a .ENTER on the appropriate channel, assign the proper completion routine, and do a .READC,channel,...,completion routine then terminate. The scheduling of a completion routine in this example is the equivalent of an ACCEPT(....) in the ADA design.

5.4.1.2 RT-11 Implementation - Monolithic -

Figure 5.12 presents the structured diagram of the implementation. In this implementation, a monolithic mainline routine handles all of the management activity on behalf of the entity. The commitment to structured design

#If each channel is tied to a dedicated completion routine, this is easy. If, however, the completion routine code is used re-enterantly, the routine must first pick up its context, then begin.
and modularity has been maintained at the entity level, with each entity still being implemented as a job under RT-11, but the actual manager implementation is strongly reminiscent of the monolithic managers of earlier chapters.

FIGURE 5.10: OSI ENTITY IMPLEMENTATION IN RT-11
TYPE entryType = RECORD;
status := (open, closed);
Pending, beingServed : queueType; END;
taskType = RECORD;
entry : ARRAY[ entryID ] of entryType;
channel : INTEGER;
state := ...; END;

VAR manager : ARRAY[ managerID ] of TASKTYPE;

PROCEDURE ManagerAsACompletionroutine;
BEGIN
  ( get manager info, channel info,
    as this is the start of completion routine)
IF manager[ channel ] ≠ NIL THEN WITH manager[ channel ] DO BEGIN
  CASE entry, status of
  open := BEGIN
    processRequest;
    DetermineNewOpenEntries;  (-- must process any entries)
    WHILE MoreEntries DO BEGIN
      (-- just opened)
      processRequest( ... );
      DetermineNewOpenEntries;
    END;
    END;  (-- can't do it now, but some)
  closed := queueRequest( ... );  (-- instance of completion routine)
  END ( CASE )
END;  (-- will do it when open)

Figure 5.11 : RT-11 COMPLETION ROUTINE OF ADA TASKING

5.4.1.3 RT-11 Implementation - Monolithic -

Completion routines are still used in this implementation, but only to pass information to the monolithic manager. The manager maintains control off all virtual connections, states, packets, etc., and performs all READ and WRITE requests on the monitor. After performing all necessary tasks, and while awaiting new processing, the manager suspends itself. When a request or notification arrives for the manager on a particular channel, the appropriate completion routine is scheduled, which deposits the message in the appropriate queue, resumes the mainline, and exits. The mainline, now resumed, processes the message.
Should a request arrive while the mainline is executing, the completion routine pre-empts it, deposits the message in the queue, and issues a resume command to the operating system. Once the mainline has finished its last function, it will execute a suspend, but since it was already resumed, will be continued to process the next message. The SUSPEND/RESUME pair act like a generalized semaphore, hence any number of resumes can be queued, and the mainline will be correctly scheduled.
5.4.1.3.1 Mapping Distributed Design to Monolithic Implementation

The design for this implementation was a highly distributed one, while the actual implementation was a monolithic routine supported by completion routines.

The mapping from the distributed design to the monolithic implementation is not difficult however. It is far more obvious than the alternate mapping. Figure 5.13 shows some mappings of distributed design mapped into a monolithic implementation. The package interface procedures map directly to the completion routines. The placing the message in the appropriate queue and setting the proper flags, then RESUME-ing the mainline corresponds to the design entry call. The message queues in the manager correspond to the entry to the design virtual circuit managers, while CONNECT_REQ, CONNECT_IND, etc queues correspond to the dispatcher entries. The procedure FREE corresponds to the FREE entry in the dispatcher, and manipulates the list of free connections, while the device handlers below correspond to the low-level routing tasks. The higher level routing tasks in the design map to the N-1 CONNECTION_INFO queues, and the internal entries between these tasks (#PDU_DATA_IND, etc.) map to the procedures connecting the N data structures with the N-1 data structures. Decisions as to which entry to accept map into the evaluation of control flags in the connection control structures.
5.4.2 RSX Implementation of an OSI node

Figure 5.14 shows a high level view of a potential RSX implementation of an OSI node. Since RSX is a true multi-tasking executive, it should be possible to map each task of the OSI design specification to an RSX task. This does not work well in practice, however, for the following reasons:

1. The task switching overhead of a fully generated (non-stripped) RSX system is large, which would seriously degenerate the system for other users
sharing the common processor; and,

2. The inter-task communication is quite primitive, restricted basically to task-oriented messages, which the destination task must expect and be prepared for, and a common set of significant event flags, which would be too small in number for such an implementation.

Note that such an implementation executing stand-alone on a stripped RSX system would be a feasible implementation, but when the system is sharing many other users as well, the operating system is unlikely to support it sufficiently.
FIGURE 5.14: POSSIBLE OSI IMPLEMENTATION UNDER RSX-11

Because of the above-mentioned limitations, the co-resident RSX-11 implementation closely resembles the RT-11 implementations just presented. In this case, because RSX-11 will support a higher level of multi-tasking, each entity is implemented as a task. The inter-layer communication is still performed by device drivers, which permit the interfaces to look like high-level I/O. AST's (Asynchronous Traps) provide the requests to each task, just the same as
the RT-11 completion routines did before. From this point on, the rest of the implementation has already been done, so it will not be done again here.

5.4.3 SUMMARY

All of the above co-resident implementations turned out to be basically monolithic systems, or forced the designer to be "tricky" to force the system to provide a level of concurrency not usual with such operating systems. To permit implementers to fully develop and test isomorphic implementations of the distributed design, Operating Systems which support Ada-like Low-Level Tasking, or kernels such as Multi-Pascal are needed, but such kernels must be friendly to the host or development operating system.

5.5 NEW SYSTEMS

All of the implementations discussed above had some major deficiency when it came to implementing the ADA designs; the iRMX and Multi-Pascal systems because they had to stand alone, and could not share the development system resources for test, or even for portions of the implementation; the RSX-11 and RT-11 implementations because they could not isomorphically implement the design, requiring instead the implementer to be "tricky" in his implementation to get the job done. There are at present no systems available at Carleton which would permit a simple, nearly isomorphic implementation of the ADA specification proposed.

As new systems are developed, and ADA tasking and inter-task communication concepts become accepted, these deficiencies should be overcome. Already Intel Corporation is working on a multi-tasking executive with sophisticated file handling capabilities, but still retaining the utility of RMX-80 and RMX-86. ADA development and test environments
will also begin appearing on the market shortly, which will permit a direct, machine produced implementation of the design, at least for verification and validation of the system.
CHAPTER 6
CONCLUSIONS

6.1 MEETING OBJECTIVES

The objective of this thesis was to show how Ada could be used as a design specification language for highly concurrent systems, specifically layered protocol systems, and how the designs could be implemented on non-Ada systems and distributed systems.

The objectives were met in that both X.25 network layer entities were specified using this technique, OSI-transport Layer entities were specified from the OSI—proposed Formal Description of the entities, and potential implementations under RMX-80, Multi-Pascal, RT-11, and RSX-11.

6.2 EVALUATION OF THE APPROACH

The methods examined go from conceptualization to design via a data flow analysis, or a formal description, such as the proposed OSI DDT. The resulting data analysis or Formal Description is converted into a structure graph representation of the design by assigning Ada objects and their interfaces to the "bubbles" and data exchanged between the bubbles. Next the Ada Definitions of the interfaces are written and the final design details of the system coded.

When designing any moderately complex system, a major portion of the design effort goes to organizing the design into its functional units, and to interfacing those functional units. The design approach presented in this thesis
gives a recipe-like methodology to this approach. Its strengths lie in the human-like interactions between the system objects, the ability to design and specifying, with relative ease, the interfaces first, and the ability to specify hardware and software elements in a similar manner. Since most moderate to large designs are developed and implemented by design teams, partitioning the design into functional units is essential to distributing the design effort. The methodology presented in this thesis is particularly appropriate to such efforts because of the ability using this approach to rigorously define the interface specifications between functional units initially.

In mapping the FDT to a design, and a design to an implementation, both isomorphic and non-isomorphic derivatives of a design were examined. The conclusion was that it was possible to derive monolithic implementations from distributed designs and distributed implementations from monolithic designs, as it was possible to map monolithic Formal Descriptions onto distributed design. The resulting non-isomorphic mappings were difficult to derive from the previous stage however, particularly when attempting to distribute a design which had been specified monolithically. The basic reason for this appeared to be that the monolithic design had hidden the natural concurrency of the entity to be implemented, and the follow-on designs and implementations must try to recover that concurrency, without the benefit of being able to examine the natural concurrency of the entity.

The use of Ada as a design specification language for layered protocol systems was examined. Because Ada is highly modular, permits the separate specification of interfaces and the module itself, allows the direct specification of concurrent processes (TASKS) in the code, and because Ada defines a highly structured, but almost human-like mechanism for the co-ordination of these tasks, it functions well as a
specification language.

The designs and implementations investigated were all hierarchical control designs, primarily because the procedural-like interfaces of Ada for both PACKAGE use and inter-TASK communication lend themselves to hierarchical methods. Other, non-hierarchical control designs are equally feasible, but as a first attempt were not appropriate, primarily because of the distortions of Ada’s procedural and entry interfaces to implement such designs. Chapter 4 showed in general terms how a designer might attempt such designs in Ada.

6.3 FUTURE WORK

As a result of this thesis, some avenues for future work lie in the design and implementation of concurrent systems. One track of study is to use the methodologies presented here to design non-hierarchical systems, as suggested in Chapter 4, to study the suitability of Ada in specifying such designs.

A second potential area for future study is the implementation of an Ada-like kernel to permit "JOBS" on existing systems (like RSX-11 or RT-11) to implement highly concurrent entities, at least for test and development.

Another possible avenue of future work is to attempt to implement a Transport Layer entity, along the lines outlined in Chapter 5. Of particular interest is the mapping of a highly concurrent design into a monolithic implementation.

A fourth area for future work is the complete specification and implementation of the Session, Transport, Network, Link and Physical layers on a highly distributed system, with each entity implemented on a dedicated processor, connected by thin wire, and rendezvous being implemented by
SEND-WAIT/RECEIVE/REPLY messages. This is of particular interest because, while common memory or common processor implementations of the Ada rendezvous mechanism can guarantee the correctness of the rendezvous by lockout of contenders, on a highly distributed system, this interface between tasks to affect a rendezvous must be done with messages, including the cancellation of the rendezvous. This can be tricky, as Sharman points out[15], hence more research in this area will demonstrate that we can extend the Ada rendezvous mechanism across thin-wire interfaces and permit the complete specification of systems in Ada, without regard to the underlying inter-task communications implementing the rendezvous.

Finally, a fifth area for future work is to attempt specifying other highly concurrent systems, such as operating systems, process control applications, etc., using the structured methods developed here and Ada as a specification language.
BIBLIOGRAPHY

1. Basic Specification of the Reference Model for Open System Interconnection
   ISO/TC97/WG1/N537, November, 1980

2. Bochmann, G.V., Development of an X.25 Implementation
   IEEE Transactions on Software Engineering, Volume SE-5, Number 5, September, 1979

   Submitted to ISO/TC97/WG1/N537 Ad Hoc Group on Formal Description Techniques, April, 1982

4. Bowen, A.A. and Buhr, R.J.A., Logical Design of Multiple Microprocessor Systems,
   Prentice-Hall, 1980

   To be published at Distributed Computing Conference, October, 1982

6. Cavers, J.K., Implementation of X.25 on a Multiple Micro-Processor System

7. Cavers, J.K. and Brown, A Technique for Process Synchronization in Monitors
   MicroComputer System Development Laboratory Technical Note, May 1978

8. Cavers, J.K., Iqbul, M.S. and Kaye, M.E., Multi-Pascal, a Support Package for Programming in
Pascal
MicroComputer System Development Laboratory Technical Note 79-58, May 1979

9. CCITT Recommendation X.25,
Document APVII Number 7-E, June 1980

10. FDT Based on Extended State Transition Models
Subgroup B of Ad Hoc Group on FDT submission to
ISO/TC97/WG1/N537, December 1981

11. Intel iRMX-80 Users Guide
Intel Corporation, 1980

interface for the PDP-11
Thesis - Faculty of Engineering, Carleton University, June, 1979


14. Reference Model for the Ada Programming Language - Proposed Standard Document,
U. S. Department of Defence, June 1980 (November 1980 Reprint)

15. Sharman, R., Retrofitting Ada's Tasking Mechanism to an Existing Language System
MicroComputer System Development Laboratory Technical Note 82-2, April 1982.

16. Yourdon and Constantine, Structured Design
Prentice-Hall, 1979
June, 1981

18. Buhr, R. J. A. System Design with Ada
Lecture Notes - Carleton University, April 82
(To be published by Prentice Hall)

MSDL Technical Note 81-5
DOC Contract Report DOC-CR-CS(81) Serial 0008

Thesis - Carleton University, April 1982