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 an inferior 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

THÈSES CANADIENNES

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 qualité inférieure.

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

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 RCUE
FAST PROTOTYPING AND VALIDATION OF COMMUNICATIONS PROTOCOLS

A MASTER'S THESIS
SUBMITTED TO THE
SCHOOL OF GRADUATE STUDIES AND RESEARCH
OF THE UNIVERSITY OF OTTAWA
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF SCIENCE
IN SYSTEMS SCIENCE

By
SHAHID H. MIR

August, 1983

© Shahid H. Mir, OTTAWA, Canada, 1983.
CONTENTS

ACKNOWLEDGEMENTS ........................................ iv
ABSTRACT ...................................................... v
LIST OF FIGURES ........................................... vi
LIST OF TABLES ............................................... viii

Chapter

1. INTRODUCTION ............................................. 1
2. SURVEY OF EXISTING TECHNIQUES FOR PROTOCOL DESIGN AND TESTING ............................................. 5
   Concept of Layered Network Architecture ................. 5
   Overview of ISO OSI Reference Model ....................... 8
   The Physical Layer ....................................... 8
   The Data Link Layer ..................................... 8
   The Network Layer ...................................... 9
   The Transport Layer .................................... 9
   The Session Layer .................................... 12
   The Presentation Layer ................................ 12
   The Application Layer ................................ 13
   Parallel Efforts by CCITT ................................ 15
   Basic Definitions and Terminology ....................... 17
   Protocol Specification Issues ............................ 18
   Examples of Protocol Design or Specification Errors ............ 20
   Current Protocol Validation Techniques ................... 21
   Protocol validation vs. Program validation ................. 22
   Reachability Analysis Based Techniques .................. 22
   Proving based techniques ................................ 27
   Symbolic Execution based techniques ..................... 29
   Software Testing Techniques and \relation to protocol Testing ................................ 30
   Protocol Validation -- Software Tools Perspective ........... 30
   Automated Validation of Protocols ........................ 31

3. PASCAL PROTOTYPES FOR TS AND CLASS-0 TP SPECIFICATIONS ............................................. 32
   Transport Service Specifications .......................... 33
   Transport Service Primitives .............................. 36
   Description of the Finite State Transducer ................ 37
   Queue Mechanism ....................................... 42
   Transport Connection Establishment: An Example .......... 43
   A Pascal Prototype for TS Specifications .................. 44
   System Used ............................................. 45
Design Assumptions and Implementation Restrictions .............. 47
RSX-11M System Directives used ................................ 49
Task Structure and Communication ................................ 53
Queue Mechanism of Implemented system ...................... 58
Synchronization Considerations: An Example Problem .............. 58
Deadlock Prevention Scheme ...................................... 59
Error Handling ....................................................... 63
High Level Structure of Modules ................................. 63
Class-0 Transport Protocol Specifications ......................... 67
State Table for Class-0 Transport Protocol Entity ................. 67
Transport Connection Establishment: An Example ................. 74
A Pascal Prototype for Class-0 TP specifications ................. 79
Design Assumptions ................................................. 84
Implementation Restrictions .................................. 87
Additional RSX-11M Directives Used ............................... 88
Window Mechanism ................................................. 91
Circular Window Queue ......................................... 92
Region Protection .................................................. 92
Task Structure and Communication ................................ 93
Error Handling ....................................................... 101
High Level Structure of Modules ................................ 103
Software Design Considerations and Quality Assurance .......... 107
Software Design Considerations ................................ 107
Software Testing ...................................................... 109

4. APPLICABILITY OF PROTOTYPES IN PROTOCOL TESTING ARCHITECTURES ................. 113

Introduction to related research in Protocol Testing Architectures .............. 113
Protocol Testing Architectures: A Survey .......................... 114
NPL Model for Protocol Testing Architectures ....................... 114
Other Testing Architectures ....................................... 117
Effective Role of Prototypes in Testing Architectures .................. 119
Application Plan ..................................................... 121
Test Plan .............................................................. 121
An Elementary Testing Configuration ................................ 123
Application Experience ............................................. 127

5. A SUMMARY, CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH ................. 136

6. REFERENCES ......................................................... 139
APPENDIX A: Description of routines of the Prototype for TS specifications (User A's side only) .......... 143

APPENDIX B: Pascal code of routines of the Prototype for TS specifications (User A's side only) .......... 152

APPENDIX C: Description of routines of the prototype for Class-0 TP specifications (User A's side only) .......... 181

APPENDIX D: Pascal code of routines of the prototype for Class-0 TP specifications (User A's side only) .......... 220
ACKNOWLEDGEMENTS

The author wishes to express his thanks and deep gratitude to Dr. Robert L. Probert for his guidance, suggestions and valuable supervision.

Thanks are also due to Mr. H. Ural for his helpful suggestions throughout the preparation of this thesis. The author also wishes to express his thanks to Mr. D. Lefebvre for his expert advice and help in the use of the RSX-11M system directives and for his programming suggestions.

The author also wishes to offer his special thanks to his wife Mrs. Naila Mir for her patience and moral support throughout the preparation of this thesis.
ABSTRACT

Recent increases in the complexity of computer networks have necessitated the use of protocols of increased complexity. The reliability of a data communications network depends heavily on the reliability of the protocols involved. Various methods for modelling and testing protocols have been proposed in the literature [9, 10, 12]. Here, we represent the ISO Transport Service specifications in the form of executable specifications. These specifications have been described as a combination of abstract data type technique and a finite state automaton approach, [22, 23]. We also represent the ISO Class-0 Transport Protocol specifications in the form of executable specifications. The Class-0 Transport Protocol specifications are described using a finite state automaton approach, [24]. The basic philosophy behind developing the executable specifications is that of Past Prototyping. A Prototype is a small version of the ultimate system. It may be an inexpensive subset of the whole system which does not necessarily have all system features but allows the user a hands-on experience to establish his requirements. A software prototype can be developed by using a high level programming language to directly implement the specifications of the intended system as executable specifications [19]. Prototypes of Transport Service specifications and Class-0 Transport Protocol specifications are developed on the PDP 11/34 under RSX-11M using a popular programming language (PASCAL). Representative test sequences were applied to the prototype for Class-0 Transport Protocol specifications and the results are summarized.
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>ISO OSI Reference Model</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>ISO Model vs CCITT's Model</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>Model of a Transport Connection</td>
<td>35</td>
</tr>
<tr>
<td>3.2</td>
<td>State Diagram of a Transducer</td>
<td>41</td>
</tr>
<tr>
<td>3.3</td>
<td>Interaction of Tasks (Transport Service specifications implementation scheme)</td>
<td>56</td>
</tr>
<tr>
<td>3.4</td>
<td>Deadlock Prevention Scheme</td>
<td>62</td>
</tr>
<tr>
<td>3.5</td>
<td>High Level Structure of modules for task TERMA</td>
<td>65</td>
</tr>
<tr>
<td>3.6</td>
<td>High Level Structure of modules for task TRNSDA</td>
<td>66</td>
</tr>
<tr>
<td>3.7</td>
<td>An In-Depth view of the Transport Connection (with Transport Protocol Entities)</td>
<td>78</td>
</tr>
<tr>
<td>3.8</td>
<td>Interaction of tasks (Transport Protocol specifications implementation scheme)</td>
<td>96</td>
</tr>
<tr>
<td>3.9</td>
<td>High Level Structure of task TERMA (TP specifications)</td>
<td>104</td>
</tr>
<tr>
<td>3.10</td>
<td>High Level structure of modules for task TPRENTA (TP specifications)</td>
<td>105</td>
</tr>
<tr>
<td>3.11</td>
<td>High Level Structure of modules of task NTRNSDA (TP specifications)</td>
<td>106</td>
</tr>
<tr>
<td>4.1</td>
<td>NPL Model for Protocol Testing Architecture</td>
<td>116</td>
</tr>
<tr>
<td>4.2</td>
<td>Model for Protocol Testing Architecture</td>
<td>118</td>
</tr>
<tr>
<td>4.3</td>
<td>An Elementary Testing Configuration for Laboratory testing of Protocols</td>
<td>126</td>
</tr>
<tr>
<td>4.4</td>
<td>Successful Transport Connection Establishment</td>
<td>129</td>
</tr>
</tbody>
</table>
4.5 Successful Transport Data Transfer... 130
4.6 Disconnection of a Transport Connection... 131
4.7 Erroneous CR TPDU sent to TPRENTB... 132
4.8 Erroneous CC TPDU sent to TPRENTB... 133
4.9 User A requests a Transport Connection but User B rejects the request... 134
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Test Examples applied to executable TS specifications</td>
<td>57</td>
</tr>
<tr>
<td>3.2(a)</td>
<td>State Table for Transport Protocol Entity, (Incoming Events from Transport Service User)</td>
<td>70</td>
</tr>
<tr>
<td>3.2(b)</td>
<td>State Table for Transport Protocol Entity, (Incoming Events from Network Entity/peer Transport Protocol Entity)</td>
<td>71</td>
</tr>
<tr>
<td>3.2(c)</td>
<td>Predicates for Class-0 Transport Protocol specifications</td>
<td>72</td>
</tr>
<tr>
<td>3.2(d)</td>
<td>Specific actions for Class-0 Transport Protocol specifications</td>
<td>73</td>
</tr>
<tr>
<td>3.3</td>
<td>Transport Service Primitives</td>
<td>81</td>
</tr>
<tr>
<td>3.4</td>
<td>Transport Protocol Data Units (TPDU's) used</td>
<td>82</td>
</tr>
<tr>
<td>3.5</td>
<td>Network Service Primitives</td>
<td>83</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

With advancements in computer communications and software engineering, the complexity of communications software is increasing very rapidly. Complex computer communications networks give rise to a complicated set of rules for exchanging data among a number of hosts. This set of rules form the communication protocols.

The increasing complexity of protocols makes their use more prone to errors, often resulting in erroneous or ambiguous states of data exchange. This necessitates the application of software validation techniques to communications software.

Testing architectures which can be used to test protocols or their prototypes at local (laboratory) or remote (client) sites have been proposed in the literature [20, 21]. It is imperative to develop prototypes of various services and protocols which can represent the specifications in the most accurate manner in a commonly used programming language.

Structuring and standardization of these services and protocols has become a major concern to experts in the communications area. International organizations like CCITT
(International Telegraphy and Telephony Consultative Committee) and ISO (International Standards Organization) have expended much time and money in this standardization process.

A very brief history of computer networks is presented in Chapter 2 leading to the concept of Layered Architecture of computer networks. The ISO Reference Model for Open System Interconnection is presented very briefly and comments are made on parallel standardization activities by CCITT. Some basic definitions related to communications networks are presented in Chapter 2. A section of this chapter deals with an introduction to the basic concepts of fast prototyping. Existing methods of formal description of protocol specifications are described \[11\]. A survey of existing protocol validation and verification techniques is presented. A section of this chapter discusses the applicability of current software development and testing concepts to validation of protocols.

Chapter 3 deals with the application of a fast prototyping approach to developing prototypes for Transport Service specifications and Class-0 Transport Protocol specifications. ISO Transport Service specifications and Class-0 Transport Protocol specifications are presented, \[22, 23, 24\]. An executable prototype for the Transport Service specifications is developed as a first step. The Transport Service specifications used for this purpose are described as a
combination of the abstract data type, technique, and finite state automaton approach, [22, 23]. Then an executable prototype is developed for Class-0 Transport Protocol specifications as a second step. Class-0 Transport Protocol specifications used for this purpose are described on a finite state automaton based approach, [24]. Experience gained by developing the prototype for Transport Service specifications is utilized in modelling the Network Service Provider consisting of two Network Entities and two queues. It is then included as the Network Service Provider for the prototypes of the Class-0 Transport Protocol specifications. Thus, the impact of the layered network architecture of ISO OSI Reference Model is also highlighted in the development of the prototype for Class-0 Transport Protocol specifications. A popular high level programming language (PASCAL) is used for development of the prototypes of the Transport Service specifications, Class-0 Transport Protocol specifications and the model of the Network Service Provider. Software design considerations and quality assurance issues are presented in section 3.5.

Chapter 4 deals with the application of fast prototypes developed in chapter 3 to related research in protocol testing. This chapter presents a brief survey of the protocol testing architectures proposed in the literature, role of fast prototypes in these architectures, the plan developed to exercise the fast prototype for Class-0 Transport Proto-
col specifications in an elementary configuration for laboratory testing of protocols and the results of that exercise.

A summary, conclusions and suggestions for further research is presented in Chapter 5.
Chapter 2

SURVEY OF EXISTING TECHNIQUES FOR PROTOCOL
DESIGN AND TESTING

This chapter presents the concept of layered network architecture, an overview of the ISO OSI Reference Model and parallel standardization efforts by CCITT. Some basic definitions are given. An introduction to the basic concepts of fast prototyping is given. A survey of the current protocol validation techniques is presented and relation of some software testing techniques to protocol testing is presented.

2.1 CONCEPT OF LAYERED NETWORK ARCHITECTURE

Earlier design of Computer Networks was mainly on an ad-hoc experimental basis[2]. There was not a well defined and precise set of guidelines. Networks were designed according to the individual organization's requirements. This led to a variety of problems when it came to connecting two different networks together. Consequently, International organizations like ISO and CCITT became involved in developing standards for Computer Networks design.

Today networks are organized as a hierarchy of layers, often denoted "layered network architecture". ISO has come
up with a layered model of a network for Open System Inter-
connection called the ISO OSI REFERENCE MODEL. This model is
defined and discussed in the section 2.2, [3].

Layering is a technique which involves defining the
structure of a network as being logically composed of a suc-
cession of layers. Each lower layer is isolated from the
higher layers. Each layer has its own set of functions to
perform and a set of services to provide to its next higher
layer. The basic idea of layering is that each layer per-
forms a set of functions (independent of the other layers)
and adds value to the services provided by its lower layers
in a manner that the total quality of transfer of data is
improved.

Taking the example of the ISO Network layer and the
Transport Layer, the Network Layer performs the following
functions:

1. Establish a Network connection with negotiation,
2. efficient routing of data,

etc.

This set of functions of the Network Layer is assumed by
the Transport Layer in performing its own set of functions
which are those necessary to bridge the gap between the ser-
vices available from the Network Layer and those to be of-
fered to the Transport Service User (i.e. Session Layer enti-
ty), [21]. The idea is that two peer layers "talk" to
eachother by using the "service" provided to them by the lower layer entities.

One main concept is that the Network Layer plus the layers under it constitute the Network Service Provider and together they provide a set of services to the Network Service user (i.e Transport Layer entity). Now, building on the same concept, when a Transport Layer is added to the Network Service Provider to enhance the quality of service, then the Transport Layer plus the layers below it form a Transport Service Provider which provides transport services to its user (i.e Session Layer entity) at the next layer above the Transport Layer.

This concept is very similar to the underlying concepts of modular programming where the internal working of a module is totally hidden from the other modules and the module performs a service for a calling module. The related idea in Software Engineering is called information hiding [19].

The ISO OSI REFERENCE MODEL consists of seven layers as shown in Fig.2.1. Layering is done according to well defined principles. Justification for a seven layered model can be found in [2; 3; 15, 25]. An overview of these layers can be found in section 2.2.

The standardization efforts of CCITT in this direction are described in section 2.3 and their relation to ISO activities is discussed.
2.2 OVERVIEW OF ISO OSI REFERENCE MODEL

The term Open System Interconnection (OSI) implies that information could be exchanged among systems, computers, networks etc. that are OPEN to one another by virtue of their mutual use of these interface standards. OPENNESS does not imply any particular implementation or some physical means of interconnection of the different entities but refers to the mutual acceptance and implementation of the ISO standards.

The ISO OSI Reference Model consists of seven layers. A brief description of each layer and the services it provides is given below. These descriptions are based on [3, 25].

2.2.1 The Physical Layer

The Physical layer provides the necessary mechanical, electrical or any other physical means of transferring data. It is THE physical link among the hosts.

2.2.2 The Data Link Layer

The Data Link layer converts an unreliable transmission channel into a reliable channel which is used by the Network layer directly above it. It is therefore providing a service to ensure reliability of transfer of data.
2.2.3 The Network Layer

The Network layer is mainly concerned with the routing of data from a source host to a destination host. This layer is responsible for effective routing i.e attempting to avoid congestion. This layer is extremely important in the sense that the efficiency of the network depends on it. The effect of poor routing could be disastrous in a point to point network but the routing problem does not arise in satellite communications where a single channel exists [2]. Packet Switching Public Data Networks (PDN's) have been established in many countries based on the CCITT Recommendation X.25 virtual circuit services. These networks are being interconnected to provide an inter-network data communications facility. CCITT has defined Recommendation X.75 for such connections. Special attention has been given to multiple circuit procedures between communicating Signal Terminating Equipment, which permit virtual calls to simultaneously exist over a number of physical circuits. This allows for load sharing and graceful recovery from circuit failures provided at least one circuit remains operational, [26]

2.2.4 The Transport Layer

The function of the Transport layer is to provide reliable host to host communication for use by the Session layer above it. It hides the details of the communication subnet-
work from the Session layer. The control of data transportation from source host to destination host in an efficient and cost effective way without letting the higher layers know about the details is the job of the Transport layer [1, 25].

More formally, the functions performed by the Transport Layer are at least those necessary to bridge the gap between the services available from the Network layer and those to be provided to the Transport Layer users i.e. a Session entity, [23].

The basic functions of the Transport Layer can be categorized as:

1. Transport Connection Establishment phase.
2. Transport Data Transfer phase.
3. Transport Connection Release phase.

A brief description of the Transport Service specifications describing the above functions is presented in section 3.1. Details can be found in [22, 23].

The Transport Protocol has five different classes. Each class defines a set of functions. These classes are listed below with their definitions. Details of each class can be found in [24].

1. Class 0: Simple class. Class 0 provides the three phases of Transport connection listed above with er-
ror detection and reporting. Recovery from these errors is not part of the Class-0 Transport Protocol specifications.

2. Class 1: Basic Error Recovery Class. This Class provides the functionality of Class 0 along with the ability to recover after a failure is signalled by the Network Service without involving the Transport Service user.

3. Class 2: Multiplexing Class. Class 2 provides multiple Transport connections with or without individual flow control. No error detection or error recovery is provided. Multiplexing and demultiplexing allows several transport connections to share a network connection at the same time. Multiplexing allows the concatenation of TPDU's belonging to different transport connections to be transferred in the same Network Data primitive.

4. Class 3: Error Recovery and Multiplexing Class. Class 3 provides the functionality of Class 2 plus the ability to recover after a failure is signalled by the Network Service without involving the Transport Service user.

5. Class 4: Error Detection and Multiplexing Class. This class provides the functionality of Class 3 plus the ability to detect and recover from lost, duplicated, or out-of-sequence Transport Protocol Data
Units (TPDU's) without involving the user of the Transport Service.

2.2.5 The Session Layer

The purpose of Session Layer is to assist in the support of the interactions between cooperating Presentation Layer entities. To do this, the Session Layer provides services which can be classified in the following two categories:

1. Binding two presentation entities into a relationship and unbinding them. This is called the Session Administration Service.

2. Control of data exchange, delimiting and synchronizing data operations between two Presentation entities. This is called the Session Dialogue Service [2].

2.2.6 The Presentation Layer

The purpose of this layer is to provide the set of services which may be selected by the Application layer (on top of it) to enable it to interpret the meaning of the data exchanged. These services are for the management of the entry exchange, display and control of structured data. In other words it performs useful transformations on the data to be sent such as text compression [1, 25].
2.2.7 The Application Layer

This is the highest layer in the ISO OSI Reference Model. Protocols of this layer directly serve the end user by providing the distributed information service appropriate to an application, to its management and to system management. Management of Open System Interconnection comprises those functions required to initiate, maintain, terminate and record data concerning the establishment of connections for data transfer among application processes. The other layers exist only to support this layer.

An application is composed of cooperating Application Processes which communicate according to Application Layer protocols. Application processes are the ultimate source and sink for exchanged data, [25].
FIG. 2.1

ISO REFERENCE MODEL FOR OPEN SYSTEM INTERCONNECTION
2.3 PARALLEL EFFORTS BY CCITT

The information in this section is based on [2]. CCITT is developing a 'Layered Model of Public Data Networks Service Applications' for the purpose of interconnecting a number of hosts or gaining access to the network services, [2]. ISO has already come up with the OSI Reference Model which has been presented in the previous section. A high degree of compatibility exists between CCITT's and ISO's models which is apparent from Fig.2.2. This compatibility is facilitated by the cooperation which already exists between the two organizations.

Both ISO and CCITT have adhered to the structuring technique of Layering. Both have come up with a seven layered model (see Fig.2.2). The major difference is in the interpretation of the services provided by the Network and Transport layers and their relationship to X.25 virtual circuits. This is also visible in the Fig.2.2. CCITT's view is that the services provided by the Transport and the Network layers are identical except in the quality of service provided. ISO maintains that the services provided by the Network layer are somewhat primitive, resulting in distinctly different services provided by the Network and Transport layers.
FIG. 2.2
ISO's MODEL VS CCITT's MODEL
2.4 BASIC DEFINITIONS AND TERMINOLOGY

1. Communication Protocols:

Communication Protocols can be defined as the set of rules governing the orderly exchange of data and messages between peer communicating entities. Thus there would be a protocol at every level or layer of the ISO model described in the previous section.

2. Service Specifications:

Layer N provides a set of services to the user Layer N+1 above it by executing Layer N protocol, which is a set of functions, using the services provided by the Layer N-1 below it. The set of services provided by the Layer N is described by the Layer N Service Specifications. Layer N+1 service is a combination of Layer N service and Layer N+1 protocol.

Service specifications for a layer can be expressed in a Formal Description Language, [14, 17, 29]. Executable Service specifications in the form of a prototype can be obtained using a programming language. The programming language should be used with extreme care and in a strictly structured way so that no errors or ambiguities are introduced in constructing the prototype. This is the method we adopted for developing prototypes for Transport Service specifications and Class-0 Transport Protocol specifications, (see Chapter 3).
3. Protocol Specifications:

The Layer N Protocol Specifications describe how the Layer N Service is provided through specific sequences of interactions between peer Layer N entities. The peer Layer N entities actually communicate using the Service provided by Layer N-1.

4. Service Primitives: Layer N Service Primitives describe the operations at the interface (access point) between the Layer N and Layer N+1 through which the Layer N Service is provided. Layer N+1 uses Layer N primitives for communication with the Layer N entity.

2.5 PROTOCOL SPECIFICATION ISSUES

Specification techniques developed for conventional software cannot be readily applied to specifying communication protocols because protocols have some time related properties [9, 10]. Protocols are implemented in software but their functionality involves a high degree of concurrency [8, 10]. It is important to note that imprecise service specifications of a layer, say layer n, can affect the services offered to layer n+1. For example if the specifications of the Transport Service are ambiguous, the Transport Layer would not be guaranteed to supply the set of services expected by the Session Layer. Therefore, to promote completeness and consistency, the service specifications of any layer should be laid down in an extremely precise and formal manner.
Formalizing requirements is the first step towards designing any piece of software. Imprecise definitions of protocol requirements give rise to imprecise, ambiguous and/or incorrect protocol specifications which result in faulty design.

Protocol errors can result from ambiguous and imprecise protocol specifications e.g Virtual Circuit establishment and Clearing in X.25 Packet level protocol as originally proposed by the CCITT, [5, 6].

Protocol Design errors can usually be attributed to the misinterpretation of the specification by the designer. Most protocol errors have been linked to erroneous design. Ambiguity in X.25 Virtual Circuit establishment state diagrams was a significant design error, [5, 6].
Protocol implementation errors arise due to incorrect programming, misinterpretation of correct design by the programmer etc. Most persistent errors found in protocols are specification and design errors.

2.6 EXAMPLES OF PROTOCOL DESIGN OR SPECIFICATION ERRORS

These examples have been put under a separate heading because the author believes that it sometimes becomes very difficult to attribute an error strictly to incorrect specifications or erroneous design. It should be noted here that the common protocol specification and design errors are usually categorized and listed as the following classes:

1. **Deadlock condition**

   This situation arises when two communicating entities wait indefinitely for the other either to complete or initiate an action. In the case of a deadlock, the entities are not exchanging any information. They typically do not consume the networking resources such as CPU time.

   A number of deadlock conditions have been identified in the literature [4].

2. **Infinite Message Looping**
An Infinite Message Looping condition occurs when two communicating entities are in a loop such that, upon receipt of a message from one entity, the other responds with a message which in turn causes the first to repeat its first message transmission. This interchange continues indefinitely[4].

The effect on the communicating entities is similar to that of deadlocks i.e. further effective communication between the two entities is halted. However, in an Infinite Message Looping condition, the entities are consuming the network resources and are therefore, more costly than deadlock conditions.

Discussion of the causes and effects of these conditions is beyond the scope of this thesis. The interested reader is referred to [4].

2.7 CURRENT PROTOCOL VALIDATION TECHNIQUES

For the sake of precision and accuracy, there is a need to express the specifications of a protocol in a Formal Description Language. These specifications can then be expressed in the form of executable specifications and can be tested for any errors or anomalies using protocol testing techniques. This section contains a survey of current Protocol validation techniques. The terms Validation and Verification are interpreted as being distinct in the context of
Software Testing technology. In this thesis the terms Protocol Validation and Protocol Verification both denote activities to ascertain that a protocol performs its function according to the set of original requirements. This activity could be the modelling and testing of the protocol by running a series of selected test cases through it and verifying it's correctness. Performance testing issues are not discussed here.

2.7.1 Protocol validation vs. Program validation

Protocol Validation techniques are quite similar to those used for Program validation, [8, 10]. Some program validation techniques have been applied successfully to protocol validation [8, 10]. To handle typical properties of protocols like concurrency, new approaches have been developed for protocol description e.g Finite State Automaton approach, Formal Description Techniques (FDT), [22, 23, 24, 29].

A formal classification of available techniques for Protocol validation is presented below. It should be noted at this point that this classification would be from the point of view of validation of protocols and not computer programs in general. Thus, the properties of protocols have been strictly kept in mind before attempting to make this classification. The major classes of protocol validation techniques are listed below with their explanation in the following subsections:
1. Reachability Analysis based techniques.
2. Formal Proving based techniques.

2.7.2 Reachability Analysis Based Techniques
Reachability analysis is based on exhaustively exploring all the possible interactions of two (or more) entities within a layer. A composite state of the system is defined as a combination of the states of the cooperating protocol entities and the lower layer connecting them. From an initial state, all possible transitions are generated leading the system model to new system states. This process is repeated for each of the newly generated states until no new states are generated. The new system states are then analysed by the designer for error. For example, if no state transition is taking place, then there is a possible deadlock situation or a termination condition. Also, if the pattern of state transition is in the form of an infinite loop, then there could be a Ping-ponging situation.

Reachability Analysis is basically a Finite State Description based validation technique. It therefore requires a finite state model description of the protocol. Following are the types of finite state models which have been used to describe a protocol:

1. Duologue Matrix Model
The protocol is modelled as a pair of interacting processes. These processes are themselves represented as directed graphs [12]. A message transmission (represented by an integer) is represented by a pair of corresponding arcs, one in each graph. The process which sends the message has a negative value of the integer along the side of its graph while the process receiving the message has a positive value of the integer along the side of its graph. A sequence of transitions within a process is called a UNILOGUE and is represented by a path within its graph. A sequence of transitions between the two processes is called a DUOLOGUE and is represented by the pair of participating unilogues and their paths. A Duologue Matrix expressed as \( D = \text{dij} \) represents all the Duologues where \( \text{dij} \) is the Duologue formed by coupling the \( i \)th unilogue of the first process with the \( j \)th unilogue in the second process.

These Duologues can then be checked for particular kinds of erroneous behavior.

Limitations of the Duologue Matrix Model are:

a) Only two processes can be considered for analysis.
b) The number of duologues increases rapidly with increasing complexity of the state diagrams being validated. Therefore, Duologue Matrix analysis is
limited to the protocols which do not contain any cycles and therefore, return to their initial state after a finite number of interactions.

2. Perturbation Model

This is an extension of the Duologue Matrix model developed by C.R.West [18]. It overcomes one of the limitations of the Duologue Model i.e it does not require the interacting processes to come back to their initial states with the same periodicity. On the other hand, it covers all possible state transitions (starting from an initial state) by perturbing the current state of one of its component processes only (i.e. all keeping all other components unchanged). Thus, new states of the whole system are achieved. This model handles many more protocols than the Duologue Model.

Example applications include CCITT Recommendations X.21 and the X.25 Packet level protocols[10].

3. Pure Finite State Model

This is a relatively primitive approach [10]. The communicating processes and the communication medium are represented by a Finite State Machine which as a whole represents the system. Transition from one state to another are triggered by internal or exter-
nal events. Validation is done on Reachability Analysis concept. This however, leads into the traversal of a massive reachability tree in case of complex protocols.

It has been used in validating the Call Set-up and Clearing procedure in X.25,[ 5, 10].

4. General Transition Model

This model combines the finite state machine approach and a programming language approach. Each communicating process is described in terms of

a) A finite state diagram.

b) A set of program variables.

A state transition is described by

c) A token indicating the transition's current position in the transition diagram.

d) The transition's effect on the system represented by a set of values of the program variables.

This combined approach allows complex models to be validated using the Reachability analysis technique and allows for software testing concepts to be used also.

It has been applied to ascertain the validity of X.25 Packet level protocol and HDLC protocol,[ 10].
2.7.3 **Proving based techniques**

A protocol, when considered as a program, can be subjected to program proving techniques [13]. For this purpose, the specifications of the protocol must be written in a very precise manner. Some of the program proving based techniques applicable to protocol validation are given below:

1. **Inductive Assertion Method**

   This technique of proving a protocol involves the attaching of statements called assertions at certain points of the program (protocol) which are required to be true. These could be values of boolean variables. Ideally, the assertions would be derived from the service specifications but since the definition of the services are not precise enough, the assertions could be formulated by the tester. The truth of an assertion follows from the truth value of the assertions at all points previous to it in the program hence the name Inductive Assertions [13]. After formulation of the assertions, it can be proved that the programs representing the protocol entity satisfy these assertions.

   The difference between program and protocol proving using this method is in the complexity of the model. Both programs and protocols can be modelled in terms of processes.
The completeness of a proof of correctness relies on the completeness of modelling of the processes and the communication medium, in having exercised all possible paths and verified every assertion \[13\].

2. **AFFIRM AND SPEX**

There is a specification language called SPEX used to write formal specifications of programs. This has been tailored to write protocol specifications \[14\]. There also exists a semi-automated verification system called AFFIRM which can be used to verify the correctness of an algorithm to realize a particular function in a computer network. An example of the application of this system to the CONNECTION ESTABLISHMENT function can be found in \[14\]. Furthermore, a connection protocol's specifications are expressed in SPEX in the same reference.

3. **GYPSY**

This is a software engineering system which provides a unified language for expressing both specifications and programs so that high level specifications can be progressively refined into detailed programs. Program modules can be both verified in advance or checked against their specifications at run time for particular inputs, \[17\].
2.7.4 **Symbolic Execution based techniques**

This approach to protocol verification is to some extent unique. Symbolic Execution is an effective program verification techniques in which the subject program is executed symbolically, i.e. the variables take on symbolic values. Symbolic values are algebraic identifiers or expressions containing symbolic as well as numeric values, [8, 27].

For the purpose of protocol verification, this technique has been related to the Perturbation method of modelling protocols [8]. This brings it essentially under the heading of Reachability Analysis based techniques. The advantages of this technique when applied to the Perturbation model are as follows:

1. The protocol is interpreted by the verifier in essentially the same way as in reality. Therefore, it is easier to describe and verify protocols involving timing assumptions, counters and behavior affected by the actual contents of the message rather than merely by its arrival, [8].

2. Use of this technique substantially reduces the semantic size of the proof tree since symbolic rather than actual values are used and they prevent the global state space to explode.

3. By using the same technique for program and protocol verification, we can verify systems comprising both kinds of specifications.
A system which verifies protocols using Symbolic Execution (in relation with Perturbation model) is described in [8, 30]. It has been implemented as part of an existing program verification system called MCS (Microprogram Certification System), [30].

2.8 SOFTWARE TESTING TECHNIQUES AND RELATION TO PROTOCOL TESTING

There exists a diversity of Software Testing and Validation techniques which can be effectively tailored to the need of Protocol Validation.

Derivation of effective test sequences for protocol testing can be as difficult as for program testing. It is possible to enhance the existing specification based (Black Box) test case design techniques by adapting some existing testing strategies such as Functional Testing, Boundary Value Analysis and Error-Based Testing to meet the peculiar properties of communications protocols, [20]. It should be noted here that the enhancements to the existing methods should not be protocol dependent.

2.8.1 Protocol Validation -- Software Tools Perspective

This concept should be applied during the protocol design and implementation. There exist software tools which provide a mechanism for continuously monitoring the development of software. An example is CEMS (Continuous Evaluating and Mo-
onitoring System) which provides a number of interfaces e.g. between the user/designer, designer/tester etc. [31]. This or any such system could be extremely useful in coming up with a better design with fewer modifications to be carried out.

2.8.2 Automated Validation of Protocols

Automated validation of protocols has been carried out using Duologue Matrix Analysis, [12], (see section 2.7.2). A re-formulated version of this theory was programmed in a system that can identify protocol errors automatically, [7]. The underlying concept is that state diagram representation of a protocol facilitates the design of a system to automatically analyze its behavior. This validation process does not establish the correctness of semantics of the interaction i.e. whether or not the executable interactions accomplish a meaningful exchange of data [7].

This process was used for validation of the Call Establishment procedure of CCITT Recommendation X.21, to demonstrate its applicability in a real environment [7].

- 31 -
Chapter 3

PASCAL PROTOTYPES FOR TS AND CLASS-0 TP
SPECIFICATIONS

This chapter presents Transport Service (TS) Specifications, Class-0 Transport Protocol (TP) Specifications and corresponding Pascal implementations in the form of prototypes. The TS Specifications described in section 3.1 are a combination of a finite state automaton model and an abstract data type model [22]. Its Pascal implementation in the form of a prototype is given in section 3.2. The Class-0 TP specifications are presented in the form of a state table in section 3.3, [24]. Its Pascal implementation in the form of a prototype is described in section 3.4. Experience gained by developing the prototype for TS specifications is utilized in modelling the Network Service Provider underlying the prototypes of the Transport Protocol Entities. Section 3.5 summarizes software design considerations and quality assurance activities.
3.1 TRANSPORT SERVICE SPECIFICATIONS

A brief description of the TS specifications is given in this section. This description is based on [22, 23] and is purposely kept brief. Detailed description can be found in [22, 23]. The TS specifications described below assume only a single transport connection between two users [22]. In practice, multiple connections would be permitted.

The functions in a transport layer are those necessary to bridge the gap between the services available from the network layer and those to be offered to the transport service users (such as a session layer). The set of transport services provided by the transport service provider (TSP) to its user consists of the services provided by all layers below it and the functions performed by the transport layer (see Fig.3.1 and Fig.3.7).

Consider the model of a Transport Connection given in Fig.3.1. There is an interaction of four entities namely two finite state Transducers and two queues. User A communicates with the transducer A through its TSAP. Transducer A communicates with its peer transducer B according to a set of specifications given in the state diagram of Fig.3.2. Transducer B on the other end receives the incoming messages from its peer transducer, interprets them and gives an indication to User B through its TSAP according to the set of specifications given in Fig.3.2. User B initiates communica-
tion with User A in a similar fashion (this will be illustrated by an example in this section). The set of specifications of Fig. 3.2 is for both transducers (considered peer entities). For the sake of concentrating on the TS specifications alone, the services provided by the lower layers are assumed to be complete and correct. We model the communication medium for data exchange as two queues, one per direction.
USER A

TRANSPORT SERVICE
ACCESS POINTS
(TSAP's)

USER B

TRANSDUCER
A

TRANSDUCER
B

QUEUE A TO B

QUEUE B TO A

TRANSPORT SERVICE
PROVIDER

FIG. 3.1

MODEL OF A TRANSPORT CONNECTION
### 3.1.1 Transport Service Primitives.

The User interacts with its Transducer by means of Transport Service Primitives given below. It is to be noted that the TS primitives given below do not have service parameters for the implementation of the prototype of TS specifications given in section 3.2.

<table>
<thead>
<tr>
<th>Transport Service Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCREQ</td>
<td>Transport Connect Request. User sends to the Transducer.</td>
</tr>
<tr>
<td>TCIND</td>
<td>Transport Connect Indication. Transducer sends to the User.</td>
</tr>
<tr>
<td>TCRES</td>
<td>Transport Connect Response. User sends to Transducer.</td>
</tr>
<tr>
<td>TCCON</td>
<td>Transport Connect Confirm. Transducer sends to the User.</td>
</tr>
<tr>
<td>TDREQ</td>
<td>Transport Disconnect Request. User sends to Transducer.</td>
</tr>
<tr>
<td>TDIND</td>
<td>Transport Disconnect Indication. Transducer sends to the User.</td>
</tr>
<tr>
<td>TNODR</td>
<td>Transport Normal Data Request. User sends to Transducer.</td>
</tr>
<tr>
<td>TNODI</td>
<td>Transport Normal Data Indication. Transducer sends to the User.</td>
</tr>
<tr>
<td>TEXDR</td>
<td>Transport Expedited Data Request. User sends to the Transducer.</td>
</tr>
<tr>
<td>TEXDI</td>
<td>Transport Expedited Data Indication. Transducer sends to the User.</td>
</tr>
</tbody>
</table>
A single letter queue item is associated with each primitive described above [22]. A table showing the input primitive to the Transducer, the corresponding queue item and an output primitive that the transducer generates after receiving that item from its peer Transducer is shown below.

<table>
<thead>
<tr>
<th>Input Primitive</th>
<th>Queue Item generated / recognized</th>
<th>Output Primitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCREQ</td>
<td>q</td>
<td>TCIND</td>
</tr>
<tr>
<td>TCRES</td>
<td>s</td>
<td>TCCON</td>
</tr>
<tr>
<td>TDREQ</td>
<td>d</td>
<td>TDIND</td>
</tr>
<tr>
<td>TNODR</td>
<td>i</td>
<td>TNODI</td>
</tr>
<tr>
<td>TEXDR</td>
<td>x</td>
<td>TEXDI</td>
</tr>
</tbody>
</table>

3.1.2 **Description of the Finite State Transducer.**

According to the state diagram of Fig.3.2 based on [22] and rewritten according to our convention, the transducer can have four states, namely:

1. **IDLE (1)**: In this state the Transducer can:
   a) receive a TCREQ from its User, write a queue item 'q' on the queue and change its state to Out-Connect Pending state or
   b) receive a queue item 'q' from its peer, give a TCIND to its User and go to In-Connect Pending state.
In this state the transducer is waiting either for its own user to initiate a Transport connection establishment procedure or to respond to the initiation of a Transport connection from the user of its peer transducer.

2. OUT-CONNECT PENDING (2): In this state, the Transducer can:
   a) receive a queue item 's' from its peer, give a TCON to its User and go to Data Transfer Ready state or
   b) receive a TDREQ from its User, write a queue item 'd' on the queue and go to Idle state or
   c) receive a queue item 'd' from its peer, give a TDIND to its user and go to Idle state.

   In this state, the transducer knows that its own user initiated the Transport connection establishment i.e. the TCREQ was Out-Going. It waits for a queue item 's' or a queue item 'd' from its peer transducer or a TDREQ from its own user.

3. IN-CONNECT PENDING (3): In this state, the Transducer can:
   a) receive a TCRES from its User and go to Data Transfer Ready state or
   b) receive a TDREQ from its User, write a 'd' on the queue and go to IDLE state or
c) receive a queue item 'd' from the queue, write a TDIND to its User and go to IDLE state

In this state, the transducer knows that the user of its peer transducer initiated the Transport connection procedure, i.e., the TCREQ was Incoming and is waiting for a TCRES or a TDREQ from its own user or a queue item 'd' from its peer transducer.

4. DATA TRANSFER READY (4): In this state, the transducer can:
   a) receive a TNODR from its User, write a queue item 'i' on the queue and stay in Data Transfer Ready state or
   b) receive a queue item 'i' from its peer, give a TNODI to its User and stay in Data Transfer Ready state or
   c) receive a TEXDR primitive from its User, write a queue item 'x' on the queue and stay in Data Transfer Ready state or
   d) receive a queue item 'x' from its peer, give a TEXDI to its User and stay in Data Transfer Ready state or
   e) receive a TDREQ from its User, write a queue item 'd' on the queue and go to Idle state or
   f) receive a queue item 'd' from its peer, give a TDIND and go to Idle state.
In this state, the transducer can process all kinds of user data requests and TDREQ from its user and the corresponding responses from the user of its peer transducer including a TDREQ which it receives in the form of a queue item 'd'.
FIG. 3.2
STATE TRANSITION DIAGRAM (TRANSUDER)
3.1.3 Queue Mechanism

There is one queue per direction as shown in Fig. 3.1. A queue represents a flow control mechanism [22]. The two queues are created by the connection establishment procedure and destroyed by the connection release procedure. Details of the adding and removing of queue items to and from the queue can be found in [22].

The specifications of the queue mechanism require Transport Expedited Data to have priority over all other queue items with the exception of the queue item 'd' associated with TDREQ, [22], but it will be shown in section 3.2 that in the prototype of TS specifications, the queues are actually under the control of the operating system and not the user. This implementation therefore, has certain restrictions which will be listed in the next section. The introduction of some limitations during implementation of specifications as a prototype is a typical phenomenon, [19]. Implementation limitations do not invalidate the usefulness of the TS specifications because this implementation is intended to test the functional behaviour of TS specifications in the form of a prototype. Performance behaviour is not under test in this study.
3.1.4 Transport Connection Establishment.....An Example

Consider the state diagram of Fig.3.2 and the Transport Connection model of Fig.3.1. Assume that both Transducer A and Transducer B are in IDLE state. The following sequence of operations takes place if User A wishes to establish a Transport Connection with User B. User B could also be the initiator of the Connection Establishment procedure with a similar sequence of operations but interchanging all A's and B's.

1. User A types in the primitive name TCREQ.
2. Transducer A reads in User A's request, writes the queue item 'q' on the queue A to B and goes to OUT CONNECT PENDING state.
3. Transducer B receives this queue item 'q' from queue A to B, gives a TCIND to User B and goes to IN CONNECT PENDING state.
4. User B types in the primitive name TCRES showing its willingness to accept the Transport Connection Request from User A.
5. Transducer B reads in User B's primitive TCRES, accordingly writes the queue item 's' on the queue B to A and goes to DATA TRANSFER READY state.
6. Transducer A reads in the queue item 's' from queue B to A, gives a TCCON (Transport Connect Confirm) primitive to User A, and goes to DATA TRANSFER READY state.
This is a simple, totally synchronized example of successful transport connection establishment. Actually, User A or User B have a choice of refusing to establish the connection at any intermediate stage during the connection establishment procedure.

More complex examples of Connection Establishment and Release shall be presented in Section 3.2.

3.2 A PASCAL PROTOTYPE FOR TS SPECIFICATIONS
This section presents the system used for the implementation, the design assumptions and implementation restrictions, the operating system directives used, a structural presentation of the various tasks and their communication, the queue mechanism implementation, task synchronization problems and a high level structural view of the task modules.

The Pascal routines for User A's side are identical to those for User B's side. Therefore, we present a description of these routines and their Pascal code for User A's side only. This would help in keeping the volume of the code small. Description of the routines for the prototype of TS specifications is given in Appendix A. Pascal code for these routines is given in Appendix B.
3.2.1 **System Used**

The TS specifications described in the section 3.1 were implemented in the form of "executable specifications" using the following system:

1. The computer used was PDP 11/34 with Digital VT100 and Lanparscope XT100 type terminals.
2. The operating system used was RSX-11M Version 3.2. This operating system was chosen because it provides certain system directives such as SEND DATA and RECEIVE DATA for intertask communication. Such a communication capability between independent tasks is critical to the implementation of the TS specifications. Explanation and examples of these and other system directives can be found throughout this section and in section 3.4. Details can be found in [32].
3. The programming language used was Swedish Pascal. This version of Pascal was used because:
   a) It permits use of external procedures,
   b) It allows access to external Fortran routines,
   c) It allows a graceful exit from the program using the HALT primitive in case of an error.

Some comments on the portability of the code are presented in section 3.5.
Pascal provides constructs such as the CASE statement which can be used to represent a state table in a procedural programming language. The label of a CASE statement can be used to represent the different states while the action to be taken in that state is represented by the logic in front of the label, e.g.

```
CASE state OF

  one : begin
    receive (* user request *)
    (* other statements *)
    end;

  two : begin
    receive (* a queue item *)
    (* other statements *)
    end;

end; (* CASE *)
```

Therefore by looking at the code, the state diagram representation of the specifications can be readily compared.
to the code. Each label shown as a state in the above example represents a block of information related to that state only. State transitions can be followed by tracing the execution path through the Pascal program. Representation of TS specifications in Pascal using the CASE construct is close to the state diagram representation of the specifications but are executable.

3.2.2 **Design Assumptions and Implementation Restrictions**

Certain design assumptions and implementation restrictions were made for developing a prototype for the TS specifications. In this study the performance behaviour of the TS specifications is not under consideration since only a prototype has been developed. Emphasis is on functional behaviour of TS specifications. Therefore, the following design assumptions can be safely made without invalidating the usefulness of the TS specifications:

1. Queue A to B and Queue B to A of Fig.3.1 are implemented as system queues. The reason for this restriction is that the RSX-11M operating system directives SEND DATA requires the address of the receiving task and creates a queue for each task (to which data is to be sent). We therefore, use these system queues as the queues in the model of the Transport Connec-
tion (see Fig.3.1). The restriction therefore is that these queues are under the system's control and no queue item has priority over the others. The receiving task is automatically made aware of the origin of the incoming data. As an extension of this design, the queues shown in Fig.3.1 can be made independent tasks which can give priority to any particular kind of data on the queues over the others.

2. The global event flags used to synchronize the tasks of the implementation of TS specifications as a prototype must be chosen such that they are not used by the operating system for its own purposes.

3. The TS specifications implemented here are assumed to support a single Transport connection between two user tasks. There is an implicit convention in the TS specifications that each TCREQ initiates a unique Transport connection. This assumption does not restrict the testing of TS specifications supporting only one transport connection at a time. No capability is provided to support multiple connections between two users.

4. Due to the preceding assumption, there is a possibility of a deadlock situation in the event of both User A and User B attempting to request a Transport connection by sending a TCREQ at the same time (i.e. both users issue a TCREQ before anyone of them is aware of
a connection establishment initiation attempted by the other. This would result in both transducers going into OUT-CONNECT PENDING state. To prevent this, a deadlock prevention scheme has been proposed and implemented. This is described in section 3.2.6.

5. We have introduced a query command called QSTAT which either user can give as a user request and his respective transducer displays the current state the transducer is in. This command is useful because it gives the user a chance to check the state of his transducer.

3.2.3 RSX-11M System Directives used.

Following are the major system directives used in our implementation of TS specifications,[32]:

1. SEND DATA

This directive is actually a Fortran system routine which can also be invoked from Pascal programs. It instructs the system to queue (FIFO) a 26 byte (13-Word) block of data for the destination task. When a local event flag is specified, the indicated event flag is set for the SENDING task. Declaration of the directive in the calling Pascal routine is as follows:

Procedure send (var tsk : integer;
var buf : buff;

- 49 -
var efn : integer;
var ids : integer) extern(fortran);

where:

a) tsk = Destination Task name.
b) buf = 26-byte block of data to be sent.
c) efn = Local event flag number.
d) ids = Directive status.

2. RECEIVE DATA

This directive requests the system to dequeue a 26 byte (13-word) block of data for the issuing task. This data block has been queued (FIFO) for the task via a SEND DATA directive. The 26-byte data block and the sender task's name are returned in an indicated 30-byte buffer, with the sender task name in the first two words.

Declaration of this directive in the calling Pascal routine is as follows:

Procedure recv (var tsk : integer;
    var buf : buff;
    var ids : integer); extern(fortran);

where:

a) tsk = Sender task name.
b) buf = 30-byte data block of received data.
c) ids = Directive status.
It should be noted here that SEND DATA and RECEIVE DATA directives will eventually be used to send and receive control primitives of the TP specifications while actual data exchange will be done using two other system directives, SEND BY REFERENCE and RECEIVE BY REFERENCE. These directives will be described in section 3.4.

3. MARK TIME

The Mark Time directive instructs the system to declare a significant event after a given time interval. The interval begins when a task issues the directive. If an event flag is specified, the flag is cleared when the directive is issued and set when the significant event occurs.

Declaration of the directive in the calling Pascal routine is as follows:

```pascal
procedure mark(var efn : integer;
               var tmg : integer;
               var tnt : integer;
               var ids : integer); extern(fortran);
```

where:

a) efn = Event flag number.
b) tmg = time interval magnitude.
c) tnt = time interval unit code (for seconds, minutes etc.).
d) ids = Directive status.

This directive is used along with the directive WAITFR to put a task to sleep for a specified amount of time. MARK instructs the system to set a specified event flag (efn) after a certain time interval. Immediately following the call to MARK directive the directive WAITFR is invoked. WAITFR directive instructs the system to stop the execution until the event flag specified in its parameter is set. Therefore, by giving the same value to the event flag in both directives (MARK and WAITFR), we can put the task to sleep. MARK would instruct the system to set the event flag after the specified time. Immediately following it, WAITFR would stop the execution until its event flag is set. After the time specified in MARK, the system sets the event flag and the task wakes up (i.e. execution starts).

A combination of these two directives can be used effectively to stop execution of a task for a specified time interval. The system overhead can therefore, be reduced in case many tasks are running simultaneously.

In addition to the above mentioned directives, two more system directives called SETEF and CLREF are used. SETEF instructs the system to set a specified global event flag. CLREF instructs the system to clear a specified global event flag. Details can be found in [32].
Use of these system directives in our implementation was mainly due to D. Lefebvre as part of a fast protocol prototyping project directed by Dr. R. L Probert of University of Ottawa.

3.2.4 Task Structure and Communication

A structural diagram showing the interactions of various tasks involved in the implementation is presented in Fig.3.3. There are four tasks running independently:

1. TERMA: This task simply reads requests typed in by the User A and sends them to Transducer A.

2. TRNSDA: This task is the Transducer A of Fig.3.1 and runs according to the specifications given in the state diagram of Fig.3.2. TRNSDA can receive data from—TERMA or from its peer Transducer, TRNSDB.

3. TERMB: This is the peer task of TERMA. TERMB is identical in form to TERMA and reads in user requests from User B.

4. TRNSDB: This is the peer Transducer of TRNSDA. TRNSDB is identical in form to TRNSDA.

A simple example of communication of the four tasks is given below in the form of a sequence of operations. Concerns related to synchronization of these tasks will be addressed later. This example is intended only to illustrate the basic mode of communication among these tasks. The example describes the Transport Connection establishment proce-
due described earlier in terms of task communication. Assume that tasks TRNSDA and TRNSDB are initially in IDLE state.

1. User A types in a TCREQ on his terminal.

2. Task TERMA reads in that request at once and uses the SEND DATA directive to send it to task TRNSDA. The SEND DATA directive has task TRNSDB's address as one of its parameters. It therefore, puts the TCREQ in the task queue A (see Fig.3.3). This queue can be thought of as being logically attached to TRNSDA.

3. Task TRNSDA reads this TCREQ from the task queue A using the RECEIVE DATA directive of RSX-11M and sends the queue item 'q' to task TRNSDB via the SEND DATA directive. TRNSDA changes its state to OUT-CONNECT PENDING.

4. Task TRNSDB reads in this request using the directive RECEIVE DATA from its task queue B, writes a TCIND to User B and changes its state to IN-CONNECT PENDING.

5. User B then types in a TCRES which is read in by the Task TERMB which sends it to the task TRNSDB using the SEND DATA directive. This TCRES is written on the task queue B.

6. Task TRNSDB receives the TCRES, sends a queue item 's' to the Task TRNSDA and changes its state to DATA TRANSFER READY.
7. Task TRNSDA receives this queue item 's' from its
task queue A, writes a TCON to User A and changes
its state to DATA TRANSFER READY state.

8. User A and User B can now exchange data using SEND
DATA and RECEIVE DATA directives and through the task
queues A and B.

Table 3.1 shows various other test interactions applied
to the prototype of TS specifications. Test examples involv-
ing strict synchronization of various requests are presented
in subsection 3.2.6.
FIG. 3.3

INTERACTION OF TASKS
(TRANSPORT SERVICE SPECIFICATIONS IMPLEMENTATION SCHEME)
<table>
<thead>
<tr>
<th>TEST CASE NO.</th>
<th>USER A INPUT</th>
<th>OUTPUT TO USER B</th>
<th>USER B INPUT</th>
<th>OUTPUT TO USER A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>TCREQ, TDREQ, QSTAT</td>
<td>TCIND, TDIND, TRNSDB IS NOW IN IDLE STATE</td>
<td>TCRES, QSTAT</td>
<td>TCIND, TRNSDA IS NOW IN IDLE STATE</td>
</tr>
<tr>
<td>2.</td>
<td>TCREQ, QSTAT</td>
<td>TCIND, TRNSDB IS NOW IN IN-CONNECT PENDING STATE</td>
<td>QSTAT, TDREQ</td>
<td>TRNSDA IS NOW IN OUT-CONNECT PENDING STATE, TDIND</td>
</tr>
<tr>
<td>3.</td>
<td>TCREQ, TDREQ, QSTAT</td>
<td>TCIND, TDIND, TRNSDB IS NOW IN IDLE STATE</td>
<td>QSTAT</td>
<td>TRNSDA IS NOW IN IDLE STATE</td>
</tr>
<tr>
<td>4.</td>
<td>TCRES, QSTAT, TNODR</td>
<td>TCIND, TRNSDB IS NOW IN DATA TRANSFER READY STATE</td>
<td>TCREQ, QSTAT, TEXDR</td>
<td>TRNSDA IS NOW IN DATA TRANSFER READY STATE, TEXDI, TDIND</td>
</tr>
</tbody>
</table>

**TABLE 3.1**

TEST EXAMPLES APPLIED TO EXECUTABLE TS SPECIFICATIONS
3.2.5 Queue Mechanism of Implemented system

The Task Structure of Fig. 3.3 is similar to the Model of Transport Connection of Fig. 3.1 except for the queue structure. The queues shown in Fig. 3.3 are system queues and are completely controlled by the host system. The SEND DATA directive queues up the data items sent to any task in a system queue reserved for that task. Any data sent to this task (no matter what source) is queued in the same queue. Therefore, these system queues are shown in Fig. 3.3 as being logically attached to the task that can receive data. If it is desired to test the executable TS specifications with arbitrary behaviour of the underlying communication medium, the queues shown in Fig. 3.1 can be made independent tasks and data flow can be controlled by those tasks as an extension of this implementation.

3.2.6 Synchronization Considerations: An Example Problem

Due to the queue mechanism of our implementation, in the absence of a synchronization strategy among the four tasks, a deadlock could arise. For example, consider the case when both User A and User B initiate a Transport Connection Establishment procedure at the same time. Suppose that both transducers are currently in IDLE state (see Fig. 3.1 and Fig. 3.2). The following sequence of events will take place:

1. Both User A and User B type in a TCREQ at the same time.
2. Since there is no synchronization, both transducer A and transducer B send a queue item 'q' to each other and change their states to OUT-CONNECT PENDING (see Fig.3.2).

3. In OUT-CONNECT PENDING state, a transducer can:
   a) receive a queue item 's' or a queue item 'd' from its peer transducer or
   b) receive a TDREQ from its own user.

4. Now, since both transducers are in OUT-CONNECT PENDING state, neither can receive a TCRES from its user and change its state to DATA TRANSFER READY. Therefore, there is a deadlock.

5. The only thing possible in this situation is to initiate a Transport Disconnection procedure from either end as a result of time-out.

3.2.6.1 Deadlock Prevention Scheme

To prevent the above deadlock situation, a deadlock prevention scheme has been implemented. This scheme uses four global event flags to synchronize the four tasks. All tasks are active but they are constrained to receive data according to the time sequence diagram of Fig.3.4.

The scheme is such that control to receive/send is swapped from side A to side B and vice versa e.g. transducer A tries to receive an incoming request (from its user or from the peer transducer) for a predefined length of time (which
could also be zero seconds). If there is an incoming request, transducer A processes it according to the state diagram of Fig.3.2. If there is no incoming request from the user or the peer transducer, it sets a global event flag which allows transducer B to receive from its task queue. Transducer B repeats the same process. Therefore, the priority to receive is swapped between transducer A and transducer B. In addition to this, tasks TERMA and TERMB are also linked with their transducers. When TERMA has sent a user request to TRNSDA, it sets an event flag which allows TRNSDA to receive User A's request. TERMA then clears its own event flag. TRNSDA receives User A's request, processes it and after a certain predefined length of time, it sets an event flag for task TERMB signalling it to send User B's request (if any) to TRNSDB. TRNSDA then sets the event flag for TRNSDB to receive and clears its own event flag. TRNSDB repeats the same procedure. Therefore, even if the two users type in a request and hit RETURN at the same time, either task TERMA or TERMB would have its event flag set. Therefore, one would be able to send its user's request to its transducer before the other one.

Now, if a TCREQ is typed in by both users at the same time, only one of the transducers will be in a position to receive because of the continuous swapping of control to receive. The transducer in receiving position (say transducer A) receives TCREQ from its user first, writes a queue item
'q' on transducer B's task queue, changes its state to OUT-CONNECT PENDING and sets an event flag signalling transducer B to receive. Transducer B receives the queue item 'q', sends a TCIND to User B, changes its state to IN CONNECT PENDING and then receives the TCREQ send initially by User B. But, Transducer B has already changed its state to IN CONNECT PENDING, therefore, it cannot accept a TCREQ from User B in this state (see Fig.3.2) and treats it as an erroneous request giving an error message to its user.

Similarly, if the two users initiate a Transport Disconnection procedure at the same time, one of the two transducers will receive the TDREQ from its user first, send a queue item 'd' to its peer transducer and change its state to IDLE. The peer transducer receives the queue item 'd', gives a TDIND to its user and changes its state to IDLE. Now it receives the TDREQ sent by its user. Since it is in IDLE state, it treats the TDREQ as an erroneous request and gives an error message to its user. Use of global event flags restricts the two users to be on the same computer.

There are no guidelines in the TS specifications for the synchronization of the various tasks. Synchronization is also an issue related to the particular operating system which is being used for the implementation of the TS specifications.
FIG. 3.4

DEADLOCK PREVENTION SCHEME

- 62 -
3.2.7 Error Handling

The implemented system treats two types of erroneous requests:

1. Those sent by the user to its Transducer and
2. Those sent by a Transducer to its peer Transducer.

The first one is dealt by flashing an error message on the user's screen which shows the erroneous request and the current state of the Transducer. The second one is dealt by sending an error queue item 'e' to the transducer which sent the erroneous queue item and printing an error message on its users screen.

3.2.8 High Level Structure of Modules

A high level flow diagram for the task TERMA is presented in Fig.3.5 and a similar diagram for the task TRNSDA (Transducer A of Fig.3.3) in Fig.3.6. Since tasks TERMB and TRNSDB (Transducer B) are identical to tasks TERMA and TRNSDA respectively, their flow diagrams are ommitted.

These diagrams give a high level view of the various procedures, system directives and functions involved and their relation to each other. The calling routine is the tail of the arrow while the called routine is the head of the arrow. All procedures and functions shown in Fig.3.5 and 3.6 are external to their calling modules. They have been coded and compiled in separate files and invoked from
the calling modules. This allows for easy modification and maintainability of the system.

The name, function, usage, input parameters, output parameters etc. of all routines shown in Fig. 3.5 and Fig. 3.6 can be found in Appendix A.

Software design considerations and software testing strategy adopted will be discussed in section 3.5.
FIG. 3.5
HIGH LEVEL STRUCTURE OF MODULES FOR TASK TERMA
FIG. 3.6
HIGH LEVEL STRUCTURE OF MODULES OF TASK TRNSDA (TRANSUDER A)
3.3 CLASS-0 TRANSPORT PROTOCOL SPECIFICATIONS

Class-0 Transport Protocol specifications presented below have been extracted from ISO document [24] which presents the specifications for all five classes of Transport Protocol. The Transport Service user interacts with the Transport Protocol Entity by Transport Service Primitives described in the previous section and the Protocol Entity interacts with the Network Service Provider below it by means of Network Service Primitives according to the state table described in Tables 3.2 (a), 3.2 (b), 3.2 (c) and 3.2 (d). An outline of the state table is given below.

3.3.1 State Table for Class-0 Transport Protocol Entity

The state table shown in Tables 3.2 (a) and 3.2 (b) is only for Class-0. This state table describes the various states the Transport Protocol Entity can take (see Fig.3.7), the action/s that should be taken in that state upon reception of an input and the resultant state, [24]. Predicates and specific actions to be taken are shown in Tables 3.2 (c) and 3.2 (d) respectively. Conventions used are as follows:

1. The Protocol Entity can have incoming and outgoing events e.g. a TCREQ is an incoming event for the Protocol Entity from the Transport Service user above it and a CR (Connect Request) is an incoming event from the Network Service Provider. Similarly, there are outgoing events from the Protocol Entity to the
Transport Service user above it such as a TCCON and to the Network Service Provider such as an NDRQ (Network Disconnect Request). These events are shown in the first column of the state table in their abbreviated form. It is worthwhile to mention here that the Transport Service Primitives described in section 3.1, contain certain parameters such as the Source Address, the Destination Address, Quality of Service parameters etc. [23, 24]. A summary of these primitives and their parameters is given in Table 3.3. There also exist standard formats for various kinds of Transport Protocol Data Units (TPDU's) for example a CR TPDU is a Connect Request TPDU. A summary of these TPDU's is given in Table 3.4. Details of their format can be found in [24].

2. All states that the Transport Protocol Entity can assume are shown in the top row of the table in their abbreviated and expanded form. In OPEN state, the Transport Protocol Entity can exchange Transport User data with its peer entity. In CLOSED state the Transport Protocol Entity is dormant i.e it is not performing any function. The other states have self explanatory names.

3. Corresponding to each state-event pair is an action or set of several actions to be taken. In addition there are are conditional actions which are written as:

- 68 -
Pl : DR
    CLOSED;
NOT Pl :
    TCIND
    WPTRESP

This can be interpreted as :

if (Pl) then
begin
    send a DR (* Disconnect Request *)
    change state to CLOSED
end;

if (NOT Pl) then
begin
    send a TCIND
    change state to WPTRESP
end

Invalid state-event combinations are left blank [24].
<table>
<thead>
<tr>
<th>EVENT</th>
<th>WAIT FOR NETWORK CONNECTION (WFNC)</th>
<th>WAIT FOR CONNECT CONFIRM (WFCC)</th>
<th>OPEN</th>
<th>WAIT FOR TRANSPORT RESPONSE (WFTRESP)</th>
<th>CLOSED</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCREQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P0: TNIND CLOSED; P2: NCREQ WFNC; P3: CR WFCC; P4: WFNC</td>
</tr>
<tr>
<td>TCRESP</td>
<td></td>
<td></td>
<td>CC OPEN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TNODR</td>
<td></td>
<td>[3]</td>
<td>OPEN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TREQ</td>
<td>[1]</td>
<td>P5: NDREQ</td>
<td>P5: NDREQ</td>
<td>DR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CLOSED</td>
<td>CLOSED</td>
<td>CLOSED</td>
<td>CLOSED</td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 3.2 (a)**

STATE TABLE FOR TRANSPORT PROTOCOL ENTITY,[24]
(INCOMING EVENTS FROM TRANSPORT SERVICE USER)
<table>
<thead>
<tr>
<th>EVENT</th>
<th>STATE</th>
<th>WAIT FOR NETWORK CONNECTION (KFC)</th>
<th>WAIT FOR CONNECT CONFIRM (KFC)</th>
<th>OPEN</th>
<th>WAIT FOR TRANSPORT RESPONSE (WFTPESP)</th>
<th>CLOSED</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCCON</td>
<td>CR</td>
<td>[1] [5]</td>
<td>[1] [5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WFCO</td>
<td>CLOSED</td>
<td>CLOSED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NRIND</td>
<td>TDIND</td>
<td>[1] [5]</td>
<td>[1] [5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[1] [5]</td>
<td>CLOSED</td>
<td>CLOSED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NDIND</td>
<td>TDIND</td>
<td>[1] [5]</td>
<td>[1] [5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[1] [5]</td>
<td>CLOSED</td>
<td>CLOSED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CR TPDU</td>
<td>TDIND</td>
<td>[1] [5]</td>
<td>[1] [5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[1] [5]</td>
<td>CLOSED</td>
<td>CLOSED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DR TPDU</td>
<td>TDIND</td>
<td>[1] [5]</td>
<td>[1] [5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[1] [5]</td>
<td>CLOSED</td>
<td>CLOSED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CC TPDU</td>
<td>TDIND</td>
<td>[1] [5]</td>
<td>[1] [5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[1] [5]</td>
<td>CLOSED</td>
<td>CLOSED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DT TPDU</td>
<td>TDIND</td>
<td>[1] [5]</td>
<td>[1] [5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[1] [5]</td>
<td>CLOSED</td>
<td>CLOSED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ER TPDU</td>
<td>TDIND</td>
<td>[1] [5]</td>
<td>[1] [5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[1] [5]</td>
<td>CLOSED</td>
<td>CLOSED</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 3.2 (b)**

STATE TABLE FOR TRANSPORT PROTOCOL ENTITY [24]

(INCOMING EVENTS FROM NETWORK ENTITY/PEER TRANSPORT PROTOCOL ENTITY)
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>T-CONNECT REQUEST UNACCEPTABLE</td>
</tr>
<tr>
<td>P1</td>
<td>UNACCEPTABLE CR TPDU</td>
</tr>
<tr>
<td>P2</td>
<td>NO NETWORK CONNECTION AVAILABLE</td>
</tr>
<tr>
<td>P3</td>
<td>NETWORK CONNECTION AVAILABLE AND OPEN</td>
</tr>
<tr>
<td>P4</td>
<td>NETWORK CONNECTION AVAILABLE AND OPEN IN PROGRESS</td>
</tr>
<tr>
<td>P5</td>
<td>CLASS IS CLASS-0 (CLASS SELECTED IN CC)</td>
</tr>
<tr>
<td>P6</td>
<td>UNACCEPTABLE CC</td>
</tr>
<tr>
<td>P8</td>
<td>ACCEPTABLE CC</td>
</tr>
</tbody>
</table>

TABLE 3.2 (c)

PREDICATES FOR CLASS-0 TP SPECIFICATIONS, [24]
<table>
<thead>
<tr>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>IF THE NETWORK CONNECTION IS NOT USED BY ANOTHER TRANSPORT CONNECTION ASSIGNED TO IT, IT MAY BE DISCONNECTED</td>
</tr>
<tr>
<td>[2]</td>
<td>RECEIPT OF AN ER TPDU (SEE SEC. 6.22 OF [24])</td>
</tr>
<tr>
<td>[5]</td>
<td>AN N-RESET RESPONSE HAS TO ISSUE ONCE FOR THE NETWORK CONNECTION IF THE NETWORK CONNECTION HAS NOT BEEN RELEASED</td>
</tr>
</tbody>
</table>

TABLE 3.2 (d)

SPECIFIC ACTIONS FOR CLASS-0 TP SPECIFICATIONS, [24]
3.3.2 **Transport Connection Establishment.... An Example**

This subsection presents an example of the Transport Connection Establishment procedure assuming:

1. There exist two protocol entities which change states and performs actions according to the state table of Tables 3.2 (a) and Table 3.2 (b) described above.

2. Every Transport Service Primitive and the TPDUs have certain parameters, (see Tables 3.3, 3.4 and 3.5), (see 24 for TPDU formats).

Consider the diagram of Fig.3.7. Suppose that both the Transport Protocol Entities are in CLOSED state, there is no Network connection available and User A wishes to establish a Transport Connection with User B. The following sequence of operations is appropriate:

1. User A types in TCREQ along with various parameters e.g the source address, the destination address and quality of service parameters[ 23, 24 ].

2. Transport Protocol Entity A receives this request, sends a NCREQ (Network Connect Request) to Network Entity A (to establish a Network Connection with its peer Network Entity B) and changes its state to WFNC.

3. The Network Entity A negotiates a Network Connection with its peer Network Entity B.

4. When this Network Connection is successfully established, i.e both Network Entities are in Data Transfer Ready state, Network Entity A sends a NCCON (Net-
work Connection Confirm) to Transport Protocol Entity A.

5. Transport Protocol Entity A which is in WFNC state, receives the NCCON from Network Entity A, sends a CR TPDU which has all of the parameters of the TCREQ primitive sent by Transport Service User A to Network Entity A and changes its state to WFCC. This CR TPDU is sent to the Network Entity A in the User Data Field of a NNODR Network Service Primitive.

6. Network Entity A receives the NNODR primitive and sends the Network Normal Data queue item to its peer Network Entity with the CR TPDU as its parameter.

7. Network Entity B receives this queue item, separates the CR TPDU from it and sends it to Transport Protocol Entity B as a NNODI Network Service Primitive. Transport Protocol Entity B is currently in CLOSED state.

8. Transport Protocol Entity B receives this CR TPDU, sends a TCIND to User B along with the information carried by the CR TPDU in its parameters and changes its state to WFTRESP.

9. User B now types in a TCRES primitive along with its parameters.

10. Transport Protocol Entity B receives this TCRES, sends a CC TPDU to Network Entity B in the User Data field of a NNODR Network Service Primitive (to be
sent to Transport Protocol Entity A) and changes its state to OPEN.

11. Network Entity B receives the CC TPDU from Transport Protocol Entity B, concatenates it to the User Data field of the Network Normal Data queue item and sends the queue item to Network Entity A.

12. Network Entity A receives this queue item, separates the CC TPDU from it and sends the CC TPDU to Transport Entity A as a parameter of the NNODI Network Service Primitive. Transport Protocol Entity A is currently in WFCC state.

13. Transport Protocol Entity A receives the CC TPDU, sends a TCCON to User A along with the information contained in the CC TPDU such as the responding address and the quality of service parameters and changes its state to OPEN.

14. Both Transport Protocol Entities are now in OPEN state i.e they have successfully established a Transport Connection and can now exchange User Data.

User B can also initiate the Connection Establishment procedure with the same sequence of operations but in the opposite direction. It should be noted here that the above example is a simplified version of Transport Connection Establishment in that both User A and User B remain willing to establish the Transport Connection during the whole process of connection establishment. In reality, either of the two
users may initiate a Transport Disconnect procedure by sending a TDREQ primitive at any stage of Transport Connection Establishment procedure.
FIG. 3.7

AN IN-DEPTH VIEW OF A TRANSPORT CONNECTION
(WITH PROTOCOL ENTITIES)
3.4 A PASCAL PROTOTYPE FOR CLASS-0 TP SPECIFICATIONS

This section presents the design assumptions made for this implementation, the implementation restrictions, the operating system directives used, structure of various tasks and their communication, window mechanism which the tasks use for sending and receiving data, error handling and a high level view of various modules. It is to be noted here that the operating system and computer used for this implementation is the same as that for the implementation of the prototype of TS specifications, namely RSX-11M on a PDP 11/34.

The impact of layered network architecture of the ISO Reference Model for Open System Interconnection is obvious in this implementation. The implementation scheme of Fig.3.8 shows a separate Transport Layer and a separate Network Layer. User A and User B can be attributed to Session Layer Entities.

A summary of the Transport Service Primitives used in this implementation along with their parameters is given in Table 3.3. A summary of the TPDU's is given in Table 3.4 and a summary of the Network Service Primitives is given in Table 3.5. It is to be noted that the Transport Service primitives and the TPDU's for Transport Expedited Data are not given in the Tables 3.3 and 3.4, because the prototype is for Class-0 TP specifications only which do not support Expedited Data option.
The Pascal routines for User A's side are identical to those for User B's side. Therefore, we present a description of these routines and their Pascal code for User A's side only. This would help in keeping the volume of the code small. Description of the routines for the prototype of TP specifications is given in Appendix C. Pascal code for these routines is given in Appendix D.
<table>
<thead>
<tr>
<th>PRIMITIVE</th>
<th>DEFINITION</th>
<th>PARAMETERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCREQ/TCIND</td>
<td>TRANSPORT CONNECT REQUEST/INDICATION</td>
<td>CALLED ADDRESS, CALLING ADDRESS, EXPEDITED DATA OPTION, QUALITY OF SERVICE.</td>
</tr>
<tr>
<td>TCRES/TCCON</td>
<td>TRANSPORT CONNECT RESPONSE/CONFIRM</td>
<td>RESPONDING ADDRESS, QUALITY OF SERVICE, EXPEDITED DATA OPTION.</td>
</tr>
<tr>
<td>TDREQ</td>
<td>TRANSPORT DISCONNECT REQUEST</td>
<td>TS USER DATA.</td>
</tr>
<tr>
<td>TDIND</td>
<td>TRANSPORT DISCONNECT INDICATION</td>
<td>DISCONNECT REASON, TS USER DATA.</td>
</tr>
<tr>
<td>TNODR</td>
<td>TRANSPORT NORMAL DATA REQUEST</td>
<td>TS USER DATA.</td>
</tr>
</tbody>
</table>

TABLE 3.3

TRANSPORT SERVICE PRIMITIVES
<table>
<thead>
<tr>
<th>TRANSPORT PROTOCOL DATA UNIT (TPDU)</th>
<th>DEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR</td>
<td>CONNECT REQUEST</td>
</tr>
<tr>
<td>DR</td>
<td>DISCONNECT REQUEST</td>
</tr>
<tr>
<td>CC</td>
<td>CONNECT CONFIRM</td>
</tr>
<tr>
<td>DT</td>
<td>NORMAL DATA</td>
</tr>
<tr>
<td>ER</td>
<td>ERROR</td>
</tr>
</tbody>
</table>

**Table 3:4**

TRANSPORT PROTOCOL DATA UNITS (TPDU's) USED
<table>
<thead>
<tr>
<th>PRIMITIVE</th>
<th>DEFINITION</th>
<th>PARAMETERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCREQ/NCIND</td>
<td>NETWORK CONNECT REQUEST/INDICATION</td>
<td></td>
</tr>
<tr>
<td>NCRES/NCCON</td>
<td>NETWORK CONNECT RESPONSE/CONFIRM</td>
<td></td>
</tr>
<tr>
<td>NDREQ/NDIND</td>
<td>NETWORK DISCONNECT REQUEST/INDICATION</td>
<td></td>
</tr>
<tr>
<td>NNODR/NNODI</td>
<td>NETWORK NORMAL DATA REQUEST/INDICATION</td>
<td>NS USER DATA</td>
</tr>
</tbody>
</table>

**TABLE 3.5**

NETWORK SERVICE PRIMITIVES
3.4.1 **Design Assumptions**

The following assumptions were made for the design of the prototype for Class-0 TP specifications:

1. The TP specifications are assumed to support only a single transport connection between two transport users. This does not in any way restrict testing of the connection establishment, transport data transfer and transport disconnection procedures of the Class-0 TP specifications between two transport users which is precisely the spirit behind developing a prototype for the Class-0 TP specifications. However, in reality for every TCREQ, there is a unique transport connection between two users.

2. The Network Layer has been modelled by two Network transducers NTRNSDA and NTRNSDB. A simple Network Connection Establishment procedure is assumed. NTRNSDA and NTRNSDB were developed by the experience gained during development of the prototype for TS specifications (see section 3.2) and they behave according to the state diagram of Fig.3.2. This is the reason why the Network Transducers are so similar to the prototype of the TS specifications. They start transferring TPDU's when they are in Data Transfer Ready state. When a Transport Protocol Entity requests a network connection from its respective Network Transducer, the transducer provides that connec-
tion with minimum negotiation with its peer network entity. Therefore, it is assumed that no error situation will occur in the Network Service Provider while establishing a network connection. This is reasonable for the purposes of building a fast prototype for Class-0 TP specifications because the concentration is on the Transport connection rather than detailed negotiation of the Network Connection.

3. It is also assumed that a single Network Connection is possible between User A's side and User B's side. This was done to have a simple Network Service Provider. Emphasis of this exercise has been given to the prototypes of the Class-0 TP specifications i.e Protocol Entities (TPRENTA and TPRENTB). Sophistication of the Network Service Provider is not an objective of this exercise.

4. It is assumed that there is a Network Normal Data queue item 'i' which has a User Data field in which it can carry a single TPDU between the Network Transducers NTRNSDA and NTRNSDB at a time.

5. The action taken by the Transport Protocol Entity in case of receipt of an ER TPDU in OPEN state is not shown in the original state table for Class-0 TP specifications, [24]. The prototype for Class-0 TP specifications handles this by sending a NDREQ to the Network Service Provider and goes to CLOSED state.
6. The receipt of any TPDU other than the ones shown in Table 3.4 is handled by the Transport Protocol Entity by sending a ER TPDU and going to CLOSED state, (see Table 3.2 (b)), [24].

7. The six tasks i.e TERMA, TPRENTA, NTRNSDA, TERMb, TPRENTB and NTRNSDB have not been synchronized because the prototype for TP specifications is meant for laboratory testing in its present form and the testers at both user sites can coordinate the user inputs to avoid deadlock situations such as the one presented in section 3.2. However, these tasks can be synchronized with the help of six global event flags so that simultaneous Transport Connect Requests or Transport Disconnect Requests by User A and User B would not lead to a deadlock or unpredictable system behaviour. It should be noted at this point that even without synchronization of the six tasks in the manner suggested above, two way simultaneous data transfer is possible.

8. A single TPDU is carried by one Network Protocol Data Unit at a time. A single TPDU carrying capability is assumed because it provides the facility to transfer TPDU's between NTRNSDA and NTRNSDB. Sophistication of the Network Layer Data transfer is not an objective here and it does not restrict testing the Class-0 TP prototype's capability to send or receive TPDU's over a simple Network Connection.
9. In case an ER TPDU is to be sent, the Protocol Entity returns the (received) invalid TPDU up to and including the 'first' erroneous parameter to its peer Protocol Entity. This has not been explicitly mentioned in the Class-0 TP specifications, [24]. Therefore, we assume that the error detection of a TPDU by the protocol entity should not proceed after the first error has been discovered.

3.4.2 Implementation Restrictions

Since the executable TP specifications is meant to be a prototype, we have not attempted to remove all implementation limitations. These limitations do not invalidate the Class-0 TP specifications in anyway and they do not restrict or limit the testing to be performed on the prototype. These implementation restrictions are:

1. The user primitives such as TCREQ, TDREQ, TCRES are contained in files with file name set to primitive name and file type set to TXT e.g. TCREQ.TXT. Each file contains the appropriate parameters for the respective primitive. This has been done so that the user can edit these files and insert the relevant parameter values. Therefore, when the user wishes to send a primitive, he only has to key in the file name and parameter information is read from that file. For primitives such as TNODR, the user is prompted
for user data from the terminal. These user primitive files must be present in the users file directory.

2. The users must always have a video terminal which is compatible with the Digital VT100 type terminal. If this is not so, then the procedures for I/O would have to be modified.

3. Both users must have different User Identification Codes (UIC's) because there are routines which have the same name for both users but have different addresses for the sender and receiver tasks.

4. User data should consist only of printable characters. Therefore, characters such as <ESC> characters are not permissible. Strings upto 118 characters are allowed. Strings longer than this will be truncated to 116 characters. The CRT terminal buffer should also be greater than or equal to 116 characters. Empty strings (only a <CR>) are allowed as user data because the user might not want to send user data even after he/she typed in TNDR.

3.4.3 Additional RSX-11M Directives Used

The executable TP specifications use the SEND DATA and RECEIVE DATA system directives which were used in the executable TS specifications also. These directives carry the control primitives of the Network Service Provider such as NCREQ, NCRES, NCIND etc. which are strings, five characters
long. To exchange data larger than 26-bytes in executable TP specifications, the following system directives are used:

1. **SEND BY REFERENCE**

   This directive instructs the system to queue FIPO a packet containing reference to a memory region (a window mapped on a region) for the destination task. Thus, permitting large blocks of data to be transferred from one task to another. This window also contains a 16-byte block that can be used to send additional information.

   Declaration of this directive in the calling Pascal routine is as follows:

   ```pascal
   Procedure sref(var tsk : integer;
                   var efn : integer;
                   var swdb : wdbb;
                   var isrb : ddd;
                   var ids : integer);extern(fortran);
   ```

   where:
   a) tsk = Destination Task name.
   b) efn = Event Flag number.
   c) swdb = A 16-byte integer array containing a Window Definition Block.
d) isrb = A 16-byte integer array containing additional information.

e) ids = Directive Status.

2. RECEIVE BY REFERENCE

This directive dequeues the next packet from the Receive by Reference queue of the receiver task. The directive exits if the Receive by Reference queue is empty, [32].

The declaration of the directive in the calling Pascal routine is as follows:

Procedure rref(var rwdb : rwdbb;
    var isrb : rdd;
    var ids : integer); extern(fortran);

where:

a) rwdb = A 16-byte integer array containing a Window Definition Block.

b) isrb = A 20-byte integer array to be used as the receive buffer.

c) ids = Directive status.

In addition to the system directives described above, three more RSX-11M directives were used namely, CRAW, CRRG, DTRG. CRAW creates an address window for a task, CRRG creates a dynamic memory region for a task and attaches the region to the task and DTRG simply detaches the attached region from a task, [32].
3.4.4 Window Mechanism

Outlines of the window mechanism and memory region allocation, attachment to a task and detachment by the task is given below. Details can be found in [32]. A window is a part of the memory region whose size and identification is defined by the programmer in the window definition block. This window definition block is sent to the receiver task in the system directive SEND BY REFERENCE. Windows were very useful for our purposes since Network Service primitives and Transport Service primitives were sent and received by reference in these windows.

Whenever a task has to send or receive by reference, it creates a memory region and attaches itself to it. All addressing is done through windows (maximum of seven) in that region. These windows can be mapped (read) or written upon.

Following are the steps a task has to go through in order to send a large amount of data by reference:

1. It creates a memory region for itself and attaches to it.
2. It creates a window in that region.
3. It puts data in that window.
4. It sends the window by reference.

The steps necessary to receive a large amount of data by reference are as follows:
1. The receiver task creates a region and attaches to it.

2. It then receives window by reference (the task is now attached to the sender's region).

3. It accesses data from the window.

4. It detaches from the region.

3.4.4.1 Circular Window Queue

The executable TP specifications employ a circular window queue which can have up to a maximum of seven windows for successive messages to be queued in the region to which the task is attached. This has been done also to prevent loss of data in the case when the sender and the receiver task change their window numbers at the same time. Therefore, when a window is sent by reference, the next window (e.g. window 2 after window 1 or window 1 after window 7) is filled with the next set of information. This circular window mechanism ensures no overlap of windows, consequently, no distortion of data.

3.4.4.2 Region Protection

Presently, there is no protection on the memory regions attached to the tasks. As an extension to this implementation, read/write privileges can be given to certain tasks to read/write on some other task's region. Details on Region Protection can be found in Section 3 of [32].
3.4.5 **Task Structure and Communication**

A structural diagram showing the interaction of various tasks is presented in Fig.3.8. There are six tasks which run independently. A brief description of these tasks is given below:

1. **TERMA** : This task continuously reads the requests from the User A and sends them to the TP Entity A (abbrev. TPRENTA).

2. **TPRENTA** : This is the Transport Protocol Entity A of Fig.3.7. It receives User A's requests and processes them according to the Class-0 TP specifications given in Tables 3.2 (a), 3.2 (b), 3.2 (c) and 3.2 (d). It also receives TPDU's from its peer entity on User B's side and processes them also according to the above mentioned Tables. This task can send and receive small amounts of data (upto 26-bytes) using the SEND DATA and RECEIVE DATA system directives and large amounts of data using SEND BY REFERENCE and RECEIVE BY REFERENCE system directives (see section 3.4.3).

3. **NTRNSDA** : This is the Network Entity A of Fig.3.7. It communicates with its peer Network Entity on User B's side and establishes a Network connection. NTRNSDA is then able to carry TPDU's in the User Data field of the Network Normal Data queue items to User B's side. These queue items when received by Network Entity B are disintegrated and the TPDU received is sent to
Protocol Entity B (abbrev. TPRENTB) as a NNODI Primitive. NTRNSDA has been developed based on the experience gained from building the prototype, for TS specifications in section 3.2. It behaves according to the state diagram of Fig.3.2. This task is capable of sending and receiving small amounts of data using the SEND DATA and RECEIVE DATA system directives and sending and receiving large amounts of data using the SEND BY REFERENCE and RECEIVE BY REFERENCE system directives (see sections of this chapter on System Directives used). It is to be noted here that NTRNSDA negotiates a Network connection with NTRNSDB first. Only then it starts carrying TPDU's to and from TPRENTA and TPRENTB.

4. TERMB: This is the peer task of TERMA and continuously reads requests from User B.

5. TPRENTB: This is the peer task of TPRENTA and performs identical functions but for User B.

6. NTRNSDB: This is the peer task of NTRNSDA and performs identical functions but for User B.

An example of communication of the six tasks is given below in point form. This example describes the Transport Connection Establishment procedure in the perspective of task communication. It is to be noted that this example does not involve the synchronization issues among various tasks as pointed out in the design assumptions. However,
these tasks can be synchronized by the use of global event flags and by giving priorities to the various tasks over the others to receive as was done in the prototype for TS specifications.
FIG. 3.8
INTERACTION OF TASKS
TRANSPORT PROTOCOL SPECIFICATIONS
IMPLEMENTATION SCHEME

- 96 -
Assume that tasks TPRENTA and TPRENTB are in CLOSED state (see Tables 3.2 (a) and 3.2 (b)) Assume that there is no Network Connection available i.e NTRNSDA and NTRNSDB are in IDLE state. Suppose that User A wishes to initiate a Transport Connection. The following sequence of events takes place, (see Fig.3.8) :

1. User A punches in TCReq. The task TERMA reads the parameters of the TCReq primitive from the file TCReq.TXT and uses the SEND BY REFERENCE directive to send this information to task TPRENTA which is written on the task queue TPREFQA.

2. TPRENTA is in CLOSED state. It receives the TCReq primitive using the RECEIVE BY REFERENCE directive. It now checks if there is a Network Connection available. There is none. Therefore, it sends a NCReq to NTRNSDA using the SEND DATA primitive which is written on the task queue NTDATQA. It then prepares a CR TPDU and changes its state to WPNC.

3. NTRNSDA receives the NCReq, sends a queue item 'q' to NTRNSDB which is written on the task queue NTDATQOB using the SEND DATA directive. It then changes its state to OUT-CONNECT PENDING.

4. NTRNSDB receives the queue item 'q' using RECEIVE DATA directive, sends a NCIND to task TPRENTB using the SEND DATA directive which is written on the task queue TPDATQOB and changes its state to IN-CONNECT PENDING.
5. Task TPRENTB is in CLOSED state. It receives the NCIND from NTRNSDB using the RECEIVE DATA directive, sends a NCRES to NTRNSDB which is written on the task queue NTDATQB using the SEND DATA directive and stays in CLOSED state.

6. Task NTRNSDB receives the NCRES from TPRENTB using the RECEIVE DATA directive and sends a queue item 's' to NTRNSDA using the SEND DATA directive which is written on the task queue NTDATQA and changes its state to DATA TRANSFER READY state.

7. Task NTRNSDA receives the queue item 's' using the RECEIVE DATA directive and sends a NCCON to TPRENTA which is written on the task queue TPDATQA indicating that a Network connection has been established. NTRNSDA then changes its state to DATA TRANSFER READY state.

8. Task TPRENTA now sends the CR TPDU to task NTRNSDA using the SEND BY REFERENCE directive. This TPDU is sent in the User Data field of the NNODR Network Service Primitive which is written on the task queue NTREFQA. Task TPRENTA then changes its state to WFCC state.

9. Task NTRNSDA receives the CR TPDU using the RECEIVE BY REFERENCE directive and concatenates the CR TPDU into the User Data field of a Network Normal Data queue item and sends the queue item to NTRNSDB using
SEND BY REFERENCE directive which is written on the task queue NTREFQ8.

10. NTRNSDB receives the Network Normal Data queue item from NTRNSDA using the RECEIVE BY REFERENCE directive and separates the CR TPDU from its User Data field. It then sends the CR TPDU to TPRENTB as a parameter of the NNODI Network Service Primitive using the SEND BY REFERENCE directive. This NNODI is written on the task queue TPREFQ8.

11. TPRENTB receives the NNODI using the RECEIVE BY REFERENCE directive. It then separates the CR TPDU from the NNODI Network Service primitive. Then, TPRENTB maps the information in the CR TPDU on a TCIND primitive and writes all of this information on User B's terminal. It then changes its state to WPTRESP.

12. User B now types in TCRES. The task TERMINB reads the information about all parameters of TCRES primitive from the file TCRES.TXT and sends it to task TPRENTB using SEND BY REFERENCE directive which is written on the task queue TPREFQ8.

13. TPRENTB receives TCRES with parameters using RECEIVE BY REFERENCE directive and, maps the information on a CC TPDU, sends it to NTRNSDB using the SEND BY REFERENCE directive in the User Data field of the Network Service Primitive NNODR. This is written on
the task queue NTREFQ0B. It then changes its state to OPEN state.

14. NTRNSDB receives the CC TPDU, concatenates it into the User Data field of the Network Normal Data queue item and sends the queue item to NTRNSDA using the SEND BY REFERENCE directive. This queue item is written on the task queue NTREFQA.

15. NTRNSDA receives this queue item, separates the CC TPDU from its User Data field and sends the CC TPDU to TPRENTA using SEND BY REFERENCE directive as a parameter of the NNODI Network Service Primitive. This primitive is written on the task queue TPREFQA.

16. TPRENTA separates the CC TPDU from the NNODI primitive and maps this information on the TCCON primitive. It then writes all of this information on User A's terminal and changes its state to OPEN state. The Transport Connection is now OPEN.

This was a simple example of Transport Connection establishment. The executable specifications actually allow any of the two users to initiate a Transport Disconnection procedure by typing in TDREQ and providing additional information for disconnection. This is handled exactly in accordance with the TP specifications given in Tables 3.2 (a), 3.2 (b), 3.2 (c) and 3.2 (d).
Once Transport Connection has been established, Transport data can be transferred on both sides using the TNODR primitive. This transfer of data is sent and received using the SEND and RECEIVE BY REFERENCE directives. The Network Normal Data primitives carry the Transport Data TPDU's (DT TPDU's) between TPRENTA and TPRENTB.

3.4.6 **Error Handling**

There are two types of errors that the Prototype handles:

1. Erroneous user primitive name. This error is handled by giving an error message to the user and prompting him for a legal primitive name.

2. Erroneous TPDU received from peer protocol entity. This error is handled by transmitting an ER TPDU to the peer TP Entity. This ER TPDU contains the erroneous TPDU up to and including the byte in which the first error occurred and the cause for rejection of the TPDU as per[24]. The ER TPDU is displayed on the terminal of the user who receives it with all its parameters in a user friendly manner. Errors in the received TPDU's are detected by the Transport Protocol Entities using the Procedure INSPTPDU.

System errors arise occasionally. The prototype has no real facilities to deal with these problems. Two types of such errors are described below:

1. **Errors arising from system directives**:

   In case of receipt of error codes from the system directives, the task halts using the HALT primitive of Swedish Pascal and has to be started again if further communication is desired with it. These errors are detectable in that the behaviour of the task is predictable. Occurrence of such an error in a prototype is tolerable because it can only disrupt the testing process and can alert the designers if they are to use the same system directive in the design and implementation of the real protocol.

2. **Implementation errors**:

   Some non-detectable errors were also discovered during the implementation process. These errors included:
   
   a) Run-time error if the keyword EXTERN was missing from the external procedure declaration but the error message would not report this error.
   
   b) Run-time error if the declared Stack or Heap size was too small. It was observed that the run-time error message did not report the real cause. The Stack or Heap size can be increased using the (/INC=nnnn) run-time option.
3.4.7 High Level Structure of Modules

Fig. 3.9, Fig. 3.10, Fig. 3.11 show the high level flow diagrams of the tasks TERMA, TPRENTA and NTRNSDA respectively. The three tasks for Side B i.e TERMB, TPRENTB and NTRNSDB are identical to those of Side A. Their flow diagrams are therefore omitted.

These diagrams give a high level view of the various procedures, functions and system directives used in each task and their relation to each other. It is to be noted here that all procedures and functions are external to their calling modules and have been separately compiled in different files.

The name, function, usage, input parameters, output parameters, exit conditions etc. of each procedure can be found in Appendix C.
FIG. 3.9
HIGH LEVEL STRUCTURE OF TASK TERMA
( TP SPECIFICATIONS)
FIG. 3.10
HIGH LEVEL STRUCTURE OF TASK TPRENTA,
( TP SPECIFICATIONS )
FIG. 3.11
HIGH LEVEL STRUCTURE OF TASK NTRNSDA
(TP SPECIFICATIONS)
3.5 **SOFTWARE DESIGN CONSIDERATIONS AND QUALITY ASSURANCE**

This section deals with the software design considerations and software testing strategy adopted during the development of the prototypes of TS specifications and TP specifications.

3.5.1 **Software Design Considerations**

The main considerations taken into account while designing the prototypes for TS and TP specifications are given below. These considerations are the same for both prototypes in principle. Therefore, they have been illustrated with reference to the prototype for TP specifications only.

**Modularity:**

Design of both prototypes is modular. For example, the prototype for TP specifications has three tasks on each user's side i.e for user A, there is TERMA, TPRENTA and NTRNSDA. The design is modular in that:

1. Each of these tasks have external procedures and functions (see Fig.3.9, Fig.3.10, Fig.3.11) for high level view of these tasks and their external routines.

2. No global variables are used for any task.
Portability:

The prototypes have been designed to promote portability as follows:

1. All procedures and functions of every task are external to the calling routines. It is assumed that the operating system on which the prototype will be used in future supports the use of external routines. A view of the procedures and functions is given in the high level flow diagrams of Fig. 3.5 and Fig. 3.6 for the prototype of TS specifications. Similar diagrams for the tasks of the prototype of TP specifications are given in Fig. 3.9, Fig. 3.10 and Fig. 3.11.

2. Whenever a task has to send or receive data using the SEND DATA, SEND BY REFERENCE, RECEIVE DATA or RECEIVE BY REFERENCE directives, it does not call any of these directives directly. Instead, it calls an external sender routine to send data using SEND DATA or SEND BY REFERENCE directives and calls an external receiver routine to receive data using RECEIVE DATA or RECEIVE BY REFERENCE directives. This makes the task portable to other operating systems since only the sender and the receiver routines would have to be modified.

The following points describe the degree of system dependence of the prototypes:
1. The display of information to the users is Digital terminal VT100 dependent.

2. The prototypes require an operating system which allows Fortran I/O from Pascal routines.

3. System directives or corresponding routines should be available for intertask communication.

The author feels that this degree of system dependence is reasonable for prototypes.

Modifiability

The most important point to note is that the prototype for TP specifications is readily modifiable. More routines can be added onto it incrementally if:

1. Multiple Transport connections are to be supported.

2. All errors in an erroneous TPDU received are to be detected.

3. Changes have to be made in the prototype for using it as the error generator for a peer entity under test (EUT) in the testing architectures (see Chapter 4).

3.5.2 Software Testing

This subsection deals with the software scheme adopted in testing prototypes of TS specifications and TP specifications. But, for the sake of brevity, the testing scheme will be presented in the perspective of the prototype of TP specifications. The software testing scheme consisted of the following:
Module testing

User A's side consists of the following tasks (see Fig.3.8):
1. TERMA.
2. TPRENTA
3. NTRNSDA

User B's side consists of the following tasks:
1. TERMB
2. TPRENTB
3. NTRNSDB

Each procedure of the tasks on User A and User B's side was compiled in separate files. The tasks on User A's side were tested individually. Then, the tasks on User B's side were tested individually according to the scheme given below. Dummy tasks are required in this scheme because the sender task requires the receiver task's name and the system directive halts the sender task if the receiver task is not found and no further testing can be performed before the sender task is started up again.

Task TERMA was tested individually with a dummy task replacing task TPRENTA. It was tested for all kinds of user requests such as TCREQ, TCRES, TNODR and TDREQ with their appropriate parameters. Then task TPRENTA was tested and tasks NTRNSDA and TERMA were replaced by dummy tasks. TPRENTA was given input from User A and responses from the
dummy task in place of NTRNSDA. The changes in its states were carefully noted by the help of debug statements and matched against the state table given in Tables 3.2 (a), 3.2 (b), 3.2 (c) and 3.2 (d). Then task NTRNSDA was tested and dummy tasks were run in place of NTRNSDB on User B's side and TPRENTA above it. It was tested for successful sending and receiving of Network primitives for establishment of a Network connection.

Tasks on User B's side were also tested individually in a similar way.

Integration testing:

Integration testing was performed in two steps:
1. Integration of modules on each user's side.
2. Integration of all modules.

Integration of modules on User A's side was done by running TERMA, TPRENTA and NTRNSDA together with a dummy task for NTRNSDB. Similarly, User B's side was tested by running TERMB, TPRENTB and NTRNSDB with a dummy task for NTRNSDA.

Then all six tasks were run together with actual user requests. At this point, a serious problem was encountered:

After having used the system directive RECEIVE BY REFERENCE, a task would not receive any data using the system directive RECEIVE DATA for an arbitrary number of times
and then suddenly receive all data using the RECEIVE DATA directive. The diagnosis of this problem was found to be a compiler error due to which system directives behave in an unpredictable manner in case RECORD type of data structures are passed as variable parameters between procedures. This problem was overcome by passing POINTERS to the RECORD's instead of the RECORD's themselves.

**Regression testing**

Regression testing was performed using the same sequence of test inputs after the problem with passing the RECORD type was overcome. This was done to determine that the effect of modifications in the tasks was local to the problem encountered and did not affect other functions of the system. This was indeed the case.

The author feels that the prototype for TP specifications developed in section 3.4 can be safely used to check the functional behaviour of the Class-0 TP specifications.
Chapter 4

APPLICABILITY OF PROTOTYPES IN PROTOCOL TESTING
ARCHITECTURES

This chapter presents an introduction to some related research in protocol testing architectures, the potential role of prototypes in these testing architectures and a plan to apply the prototypes of Class-0 TP specifications (TPRENTA and TPRENTB) in an elementary configuration for laboratory testing of protocols which is also proposed in this chapter. The results of the application of selected test sequences to the prototype TPRENTB are also presented. Some of the protocol testing concepts in this chapter have been proposed by H. Ural and R. L Probert [20].

4.1 INTRODUCTION TO RELATED RESEARCH IN PROTOCOL TESTING ARCHITECTURES

This section presents a brief survey of the models for protocol testing architectures proposed in the literature and the effective use of the prototype for Transport Protocol in these architectures.
4.1.1 Protocol Testing Architectures...A Survey

Several models for protocol testing architectures have been proposed in the literature. Two such models are presented below:

4.1.1.1 NPL Model for Protocol Testing Architectures

The National Physical Laboratory's Model for Protocol Testing Architectures provides the facility to test a Layer N protocol implementation between two open systems, one located at the Client's Site and the other at the Assessment or Laboratory Site, [21], (see Fig.4.1). The Assessment Site consists of the following components:

1. A Test Driver.
2. Layer N protocol implementation + error Encoder/Decoder.
3. Layer N-1 protocol implementation + Exception generator

The Client's Site system consists of a set of similar components as follows:

1. A Test Responder.

There is an underlying Layer N-2 communication channel common to the Client's Site and the Assessment Site. Details of the above components can be found in [21].

- 114 -
The NPL Model assumes that to test a Layer N protocol implementation, only the protocols and services below Layer N have been implemented and not those above it [21]. This architecture also considers the Implementation Under Test (IUT) to be a black box i.e. the details of the internal structure of the IUT are not considered when test sequences are selected.

The NPL model has been adopted by several research groups in different countries to develop test architectures suited to their particular needs for certification purposes, [21].
FIG. 4.1
NPL MODEL FOR PROTOCOL TESTING ARCHITECTURE
4.1.1.2 Other Testing Architectures

A model based on two open systems interacting over a network connection has been proposed in [20]. This model is shown in Fig.4.2. The basic components of this model are:

1. A Laboratory Site Test Manager.
2. A Laboratory Site Test Supervisor.
3. An Emulator for peer of entity under test.
4. A communication medium.
5. The Entity Under Test (EUT) on the Client Site.
6. The Client Site Test Supervisor.

Details of these components and their configuration for various test architectures can be found in [20].

Logical architectures for protocol testing have been developed for Laboratory (Assessment site) testing and for Remote (Client's Site) testing by expanding the model incrementally. The Laboratory Testing Architecture needs an emulator for the communication medium and the Entity Under Test is in the laboratory, whereas in remote testing the Entity Under Test is at the Client's Site and is tested over a real communication medium, [20].

The effectiveness of the black-box testing approach has been enhanced in this model by incorporating adaptations of some testing strategies such as Functional testing, Boundary-Value Analysis and Error-Based Testing, [20].
4.1.2 Effective Role of Prototypes in Testing Architectures

All of the testing architectures described above have a component called an Entity Under Test (EUT) or an Implementation Under Test (IUT). This component can be:

1. An actual protocol implementation or
2. Executable Specifications of Layer N protocol in the form of a prototype.

The Entity Under Test can be either an actual implementation of the protocol when an existing protocol implementation is to be tested or it can be a new implementation to be tested. In both cases, the protocol has been implemented. If basic design errors are detected, a lot of effort and money could be expended in redesigning and recoding the software.

This leads us to the concept of high-level software testing, which calls for detecting errors at an earlier stage of the development process i.e. before the actual implementation. This concept involves the migration of software testing methodologies from the implementation phase upward into the design and specification phases, [33].

An inexpensive way of testing the desirability of the ultimate product (e.g. a protocol implementation) is to develop a fast prototype which has all of the functional features of the specifications. As suggested by H. Ural and R. L Pro-
bert in discussions, a prototype can be the laboratory site emulator for peer layer under test. It has also been suggested during these discussions that use of prototypes at laboratory site and at the client's site allows "live" testing of specifications.

The fast prototype developed for Class-0 TP specifications (on User B's side) is used as the Entity Under Test (EUT) for running selected test sequences through it and its behaviour is observed. Changes are made to the prototype for User A so that it is possible to generate erroneous TPDUs. It should be noted that the selection of test sequences is specification-based. By applying these selected test sequences to the fast prototype, suitability of the TP Specifications can be evaluated. The protocol can be actually implemented once confidence has been developed in the prototype.

It was suggested by H. Ural during discussions that fast prototyping approach can be useful in testing the design of other components of the testing architecture such as a Test Supervisor or a Test Manager prior to their actual implementation.
4.2 APPLICATION PLAN

This section presents the test plan, an elementary configuration for laboratory testing of protocols and the resulting impact of this configuration on one of the prototypes for TP specifications. This impact involves changing one of the Transport Protocol Entities to generate TPDUs in error.

Protocol Entity A (task TPRENTA) will be changed such that it can generate erroneous TPDUs. Protocol Entity B (task TPRENTB) will be considered the Entity Under Test (EUT). Erroneous Transport Service primitives will be supplied by the tester on User B's side while erroneous TPDUs can be supplied by the modified protocol entity TPRENTA.

4.2.1 Test Plan

The test plan has three objectives as follows:

1. To detect functional errors of the protocol entity.

The Transport protocol entity performs the three main functions given below:

a) Transport Connection Establishment with negotiation between the Transport Service users.

b) Data Transfer between the Transport Service users.

c) Connection Release initiated by either of the Transport Service users or the Network Service Provider.
Each of the above functions has some sub-functions to perform such as concatenation and separation of TPDUs, mapping various parameters of user primitives such as TCREQ onto the TPDUs. The selected test sequences test these sub-functions as well.

2. To allow for functional dependencies between protocol functions.

Functional dependencies will be taken into consideration before applying the test sequences, [34]. The functions of Transport Protocol shown above are dependent on each other. For example, for successful Transport Data Transfer, successful Transport Connection Establishment is necessary and so on. Therefore, we will apply those test sequences to the Transport Protocol Entity which help establish correctness of the Transport Connection Establishment function first. Then we can check the correctness of Transport Data Transfer function of the protocol. This is followed by checking the Transport Disconnection Procedure.

3. To test error detection/reporting capabilities of protocol entity.

Class-0 Transport Protocol specifications do not require the protocol to recover from protocol errors but error detection and reporting is required. Test
sequences were selected to test the above mentioned error handling capabilities of Transport Protocol Entity. The Transport Protocol Entity is subjected to user requests and TPDU's with invalid parameter codes or values according to the testing configuration in the next subsection.

4.2.2 An Elementary Testing Configuration

An elementary testing configuration was used to show the applicability of fast prototypes to Laboratory testing of protocols. This configuration is shown in Fig.4.3. Functionally, this diagram is the same as that of Fig.3.8. The details of task queues and actual flow of data have been omitted for the sake of clarity and to highlight the features of this diagram as being an elementary testing configuration for laboratory testing of protocols. It is also shown in this section how these prototypes can be improved and incorporate them in a more sophisticated protocol testing architecture. However, this is beyond the scope of this thesis.

The system shown in Fig.3.8 does not allow User A or User B to directly input to the Protocol Entities (TPRENTA and TPRENTB). In order to enable TPRENTA to generate erroneous TPDU's on demand, it has been modified so that it can accept input from User A's side to generate errors in TPDU's (see Fig.4.3) which can then be sent to the Entity Under Test (TPRENTB). Example test sequences applied to TPRENTB are
presented. in Figs. 4.4 through 4.9. and are discussed in section 4.3.

During testing, the tester has a list of the various fields of different TPDU's. All fields on the list are numbered. The Transport Entity A in the modified form prompts the tester to:

1. Enter the field number of the particular TPDU to be changed.

2. After the tester enters the field number, he/she is prompted for the erroneous value to be entered in this field.

Once the above two steps have been done, the tester can type in 999 to signal TPRENTA to send the erroneous TPDU. The behaviour of TPRENTB in response to this erroneous TPDU can then be observed at the video terminal at which it is running. The tester can also input erroneous user primitives from User B's side and can observe the response of TPRENTB at the terminal on which the task TPRENTB is running. This scheme is simple and it provides the tester with a full capability to input erroneous parameter values and codes in the TPDU's without significantly modifying TPRENTA. The disadvantage is that the testing process is manual. Test sequences have to be input one at a time and the results noted visibly. Also, the tester has to be present at both User A's side as well as on User B's side.
As an alternative testing configuration, this process can be semi-automated by augmenting TPRENTA so that it generates erroneous TPDU's when the tester specifies such an action from User A's side. The erroneous values can be stored in a table. The tester can specify the index to the erroneous parameter value and the type of TPDU e.g. CR TPDU and the procedure preparing TPDU's (procedure PREPTPDU of task TPRENTA) can then insert erroneous values into the TPDU. Further automation can be achieved by replacing some of the functions of the human tester by a Test Manager and Test Supervisor programs, [33]. A sophisticated test sequence results reporting scheme can also be implemented. Therefore, the testing configuration of Fig.4.3 can be built upon in an incremental fashion resulting in an implementation of a standard protocol testing architecture for laboratory testing of protocols, [20, 21].
NOTE: THIS DIAGRAM IS FUNCTIONALLY THE SAME AS FIG. 3.8. DETAILS OF TASK QUEUES ARE OMMITTED FOR THE SAKE OF CLARITY.

FIG. 4.3
AN ELEMENTARY TESTING CONFIGURATION FOR LABORATORY TESTING OF PROTOCOLS
4.3 APPLICATION EXPERIENCE

Test Sequences derived on the basis of the Test Plan of section 4.2 were applied to TPRENTB. The results are summarized in Figs.4.4 through 4.9. The following points should be noted prior to reading these results:

1. Network Transducers NTRNSDA and NTRNSDB of Fig.3.8 have been combined as one Network Service Provider in the testing configuration. It is acceptable to provide only a high-level prototype Network Service Provider since our detailed focus is on the behaviour of the prototypes of Class-0 TP specifications.

2. Test sequences representing only the basic functions of the TP specifications are presented. These results clearly show the applicability of the prototype in an elementary configuration for laboratory testing of protocols. Many more sequences were run but they will not be presented here to avoid repetition.

3. Parameters of the Transport Service primitives, the Network Service primitives and the TPDU's have not been shown in tabulation of the results for the sake of clarity.

4. The results given in Figs.4.4 through 4.9 had to be manually tabulated because the RSX-11M operating system V3.2 on the PDP 11/34 distorts the contents of an output file if two or more tasks write on the same output file simultaneously. Therefore, two tasks
could not be made to write the results of a test sequence on the same output file. This problem can be resolved by writing a sophisticated File Locking Driver program. This program would instruct a task to wait until another task has finished writing on an output file if the same file is to be used as an output file for both tasks. The file locking driver was omitted because the results could be tabulated legibly after making observations and the implied time to construct the driver could not be cost-justified.

5. The fact that a TPDU is acceptable or not to TPRENTB is indicated by a note saying 'accepts' or 'rejects'.

6. The state that TPRENTB attains after sending / receiving a particular primitive, is written against that primitive e.g '.WFCC.' or '.WFNC.'.

7. The time scale is shown horizontally.
TEST SEQUENCE NO.1

USER A

TPRENTA

NETWORK SERVICE PROVIDER

TPRENTB

USER B

TCREQ

NCREQ

NCIND

NCRES

OPEN

TCInd

Accepts

ACCEPSTS CR TPDU

NNODI(CC TPDU)

NNODI(CC TPDU)

TIME

FIG. 4.4

SUCCESSFUL TRANSPORT CONNECTION ESTABLISHMENT
FIG. 4.5

SUCCESSFUL TRANSPORT DATA TRANSFER
(ASSUMING A TRANSPORT CONNECTION HAS BEEN ESTABLISHED)
TEST SEQUENCE NO. 3

USER A

TPRENTA

NETWORK SERVICE PROVIDER

TPRENTB

USER B

TREQ

NDREQ

NDIND

TDIND

CLOSED.

TIME

FIG. 4.6
DISCONNECTION OF A TRANSPORT CONNECTION
(ASSUMING A TRANSPORT CONNECTION HAS BEEN ESTABLISHED)
TEST SEQUENCE NO. 4

USER A

TCREQ

TPRENTA

NCREQ

NCON

NNODR (CR TPDU)

NDREQ

NETWORK SERVICE PROVIDER

NCIND

NCON

NNODI (DR TPDU)

NDIND

TPRENTB

NCRES

REJECTS CR TPDU

NNODR (DR TPDU)

CLOSED.

USER B

TIME

(*) ERROR IS INTRODUCED IN CR TPDU AT THIS POINT

FIG. 4.7

ERRONEOUS CR TPDU SENT TO TPRENTB

- 132 -
TEST SEQUENCE NO. 5

USERA

TPRENTA

NETWORK SERVICE PROVIDER

TPRENTB

USERB

TIME

(*) ERROR IS INTRODUCED IN CC TPDU AT THIS POINT.

FIG. 4.8

ERRONEOUS CC TPDU SENT TO TPRENTB

- 133 -
FIG. 4.9

USER A REQUESTS A TRANSPORT CONNECTION BUT USER B REJECTS THE REQUEST

- 134 -
To summarize the experience with running the test sequences, the author feels that the prototype is easy to use for running the test sequences. The results of these sequences can be readily observed on task TPRENTB's terminal and tabulated. The messages to the tester are quite user friendly. Preparation of the test sequences involves some time and effort if done manually but this process would soon be automated due to a project on automatic test sequence generation using PROLOG conducted by Dr. R. L Probert and Mr. H. Ural of the Computer Science Department, University of Ottawa. All test sequences involving the basic functions of the Transport Protocol Entities i.e. Transport Connection Establishment, Transport Data Transfer and Transport Disconnection succeeded. Those related to TPDU error detection and reporting also succeeded. It is to be noted here that two way simultaneous data transfer is possible using the prototypes for TP specifications for as long as the two users desire. Only representative test sequences for the basic functions are presented in the Figs 4.4 through 4.9.
Chapter 5
A SUMMARY, CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH

The concept of layered network architecture was presented. ISO Reference Model for Open System Interconnection was briefly presented. An introduction to basic concepts of fast prototyping was presented. The existing techniques for protocol testing were summarized. The TS specifications were presented briefly and were implemented in the form of a fast prototype in Pascal. Class-0 TP Specifications were briefly presented and were implemented as a fast prototype in Pascal. Experience gained in developing the prototype for TS specifications was utilized in building the Network Service Provider underlying the Transport Protocol Entities. Impact of the layered architecture of ISO OSI Reference Model can be seen in the implementation scheme of the prototype for Class-0 TP specifications. A brief survey of the models for protocol testing architectures was presented. Usefulness of the prototype for TP specifications was shown in an elementary configuration for laboratory testing of communications protocols. This was done by applying test sequences to the prototype to validate the communications requirements of the Class-0 TP specifications.
A prototype of a set of specifications is faster and more economical to develop compared to the real system. In some cases the ratio of time involved in building a prototype for a set of specifications versus building the real system has been 1:21, [19]. In addition to the time savings involved, the user can reasonably ascertain the correctness of his/her specifications by performing tests on the prototype before actually building the real system. It took us approximately 600 man hours to build the prototype for Class-0 TP specifications whereas in the authors' view it would take at least four times the length of time to develop a full blown implementation of the specifications. Degree of detail in building prototypes should be carefully controlled because the essence of building a prototype is to test the basic functions of a future real system. If too much detail is put in the design of a prototype, then the purpose of building a prototype would be defeated and the result would be a real system. The cost-effective use of prototyping techniques in the development and validation of communications protocols should be further investigated.

It was shown in Chapter 4 that realistic experimentation can be performed on a prototype to validate the communication requirements of the TP specifications. The main advantage of building a prototype in a popular high level programming language (e.g., Pascal) is that most systems support the language and portability of the prototype would be enhanced.
Following are some possible extensions of the use of the prototype for Class-0 TP specifications developed in this exercise:

1. Although only the Transport layer was considered in this exercise, a similar prototyping approach may be applied to the development and validation of any layer.

2. The prototype of TP specifications can be improved incrementally to be used in standard architectures for laboratory testing of protocols. Task TPRENTA (Transport Protocol Entity A) can be augmented to generate erroneous TPDU's semi-automatically. This can be done by storing the erroneous parameter codes and values in a table. The tester can specify the index to the erroneous value and the TPDU type e.g. CR TPDU and the augmented TPRENTA can introduce this erroneous value in the TPDU.

3. The prototype of TP specifications can be enhanced to be used in standard architectures for remote testing of protocols by using an existing network such as DATAPAC. However, substantial low-level modifications will be necessary to the underlying event flag mechanism.
Chapter 6
REFERENCES


[10] Software Reliability Research Group, Univ. of Ottawa 'A study to determine the techniques and tools for Validation of Communications Protocols' Report, Feb. 1982


[22] Logripppo L 'Specification of Transport Service using Finite-State Transducers and Abstract Data Types'


[26] Weir D. F, Holmblad J. B, Rothberg A. C 'An X.75 Based Network Architecture' GTE Telenet Communications Corp. USA.


APPENDIX A

DESCRIPTION OF ROUTINES
OF THE PROTOTYPE FOR TS
SPECIFICATIONS
(User A's side only)
BUFFCLEAR.PAS

1 (%%-#)
2 (**************************************************************************************************)
3
4 NAME: BUFFCLEAR
5
6 FUNCTION: To clear the event flag corresponding to the specified integer value.
7
8 USAGE: procedure buffclear(efn : efn);
9
10 buffclear(efn);
11
12 INPUT PARAMETERS: efn - the event flag number.
13
14 OUTPUT PARAMETERS: NONE
15
16 MODULES CALLED:
17   >halt
18   writeln
19   clref
20
21 EXIT CONDITIONS:
22   If the recode from the CLEF# Directive is less
23   than zero then an error message is printed and
24   the event flag was not have been cleared and
25   the processor is halted using the 'halt' primitive.
26
27 REMARKS: See RSX-11M V3.2 Executive Directives Manual on Page 6-26 for
28   more information.
29
30 **************************************************************************************************
NAME: BUFFGO

FUNCTION: To set the event flag corresponding to the passed integer value.

USAGE: procedure buffgo(efn : efin);

buffgo(efn);

INPUT PARAMETERS: efin - the event flag number.

OUTPUT PARAMETERS: NONE

MODULES CALLED: 

>halt
>writeln
>seie

EXIT CONDITIONS: Normal exit conditions.
if the recode from the SETEF$ Directive is less
than zero then an error message is printed and the
event flag may not be set and the processor is halted
using the 'halt' primitive.

REMARKS:

See RSX-11M V3.2 Executive Directives Manual on Page 6-123 for
more information.

Also see chapter 3 of the Manual for more info. on event flags and
their use.

*****************************************************************************
NAME: BUFFWAIT

FUNCTION: To wait until the event flag corresponding to the passed integer value is set.

USAGE: Procedure buffwait(efn : efn);

buffwait(efn);

INPUT PARAMETERS: efn - the event flag number.

OUTPUT PARAMETERS: NONE

MODULES CALLED: >halt
                   >writein
                   waitfr

EXIT CONDITIONS: Normal exit conditions.

if the recode from the WISE$ Directive is less
than zero then an error message is printed and
the processor is halted using the 'halt' primitive.

REMARKS: See RSX-11M V3,2 Executive Directives Manual on page 6-168 for more
information.

One should be aware of the different types of event flags that
can be used for different purposes since for example if global
event flags are used for controlling events in a particular task
then one must make sure that these flags are not used somewhere
else in the system (see chapter 3 of Manual).
**NAME:** READERM

**FUNCTION:** To read in a string from standard input and record its length.

**USAGE:**

```
procedure request(var buffer: request; var len: integer) external;
```

```
readerm(buffer, len);
```

**INPUT PARAMETERS:** NONE

**OUTPUT PARAMETERS:**
- `buffer` - the string buffer.
- `len` - the length of the string buffer.

**MODULES CALLED:** `break`, `ealn`, `read`, `readln`, `writeln`

**EXIT CONDITIONS:** Normal condition.

**REMARKS:**

The inputed line is truncated to REALLEN; so care should be taken not to exceed this value.

It is not recommended to input non-printable characters using this module since unpredictable behavior can result when these strings are outputted.
NAME: RECA

FUNCTION: To receive some information sent to the TRNSDA task by using the receive data RECEIV$ directive and to return the status of the call.

USAGE:  procedure reca(var rbuf : buff;
var ids : idss); extern;

reca(rbuf, ids);

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: rbuf  - the returned buffer
                ids  - the returned status of the directive call.

MODULES CALLED: (The symbol 'U' designates a primitive)

>ord
>writeln
recaiv

EXIT CONDITIONS:

The return code of RECEIV$ is set to greater or equal to O(zero) in case of successful completion and to less than O in case of unsuccessful completion.

For more information on the use of this directive and its return codes, see the RSX-11M V3.2 Executive Reference Manual.

REMARKS: NONE
NAME: TERMA (MAIN)

FUNCTION: This module is used to continuously READ information from USER A (TERMINAL A) and to send this information to the TRNSDA task. This information is obtained from terminal input in the form of user primitive strings.

USAGE: from MCR : 'RUN TERMA /TASK-TERMA'

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: NONE

MODULES CALLED: (The symbol "\" designates a primitive)
>halt
>read
>writeln
buffclear
buffdo
buffwait
readterm
.send

EXIT CONDITIONS: This program has been designed to run indefinitely until the task into which it runs is specifically aborted with the MCR command 'ABORT TERMA'.

If any errors are detected by one of the routines called, then the routine itself will print an error message and halt the processor. The exception to this is the call to the SDAT 'send' routine. In this case TERMA checks the return status and in case of error prints out an error message and then halts the processor by using the 'halt' primitive.

REMARKS: NONE
NAME: TRNSDA (MAIN)

FUNCTION: Transducer for side 'A'.

USAGE: from HCR : "RUN TRNSDA /TASK-TRNSDA"

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: NONE

MODULES CALLED: (The symbol '>' designates a Primitive)

>halt
>write
>writleln
>bufclear
>bufddo
>bufwait
>mark
>reca
>woab
>wnerma

EXIT CONDITIONS:
This program has been designed to run indefinitely until the task into which it runs is specifically aborted with the HCR command 'ABORT TRNSDA'.

If any errors are detected by one of the routines called, then the routine itself will print an error message and halt the processor by using the 'halt' primitive. The exception to this is the RECA routine. TRNSDA checks the status of the call to RECA and if there is an error condition then an error message is printed and the processor is halted by TRNSDA.

REMARKS:
See Ref. (22) for more details.
NAME: WOAB
FUNCTION: To put symbol into the A-B queue.
USAGE: procedure woab(messageXa : transXresp); external

woab(messageXa);

INPUT PARAMETERS: messageXa - the symbol to be sent to the queue.
OUTPUT PARAMETERS: NONE

MODULES CALLED: (The symbol '*' designates a primitive)

halt
writeIn
send

EXIT CONDITIONS:
If there are any errors detected in calling SDAT$('send') then an error message identifying the error and error code is printed onto standard I/O and the processor is then halted by using the 'halt' primitive.
The return code of SDAT$ is set to greater than or equal to
0 (zero) in case of successful completion and less than
0 in case of unsuccessful completion.

For more information on the use of these system routines
and their return codes, see the RSX-11M V3.2 Executive

REMARKS: NONE
NAME:  WJERMA

FUNCTION:  To write a TRNSDA error message on the terminal in case
illegal requests are made by TRNSDB or terminal A or to
write a query status message or one of the following
messages (successful):  (com) bina (r) kind (i) mod (r) levi

USAGE:  procedure WJERMA(messno : integer);
        statek, : state;
        reqt : request); extern

INPUT PARAMETERS:  messno  - the type of message.
statek  - the state of TRNSDA.
reqt  - the inputed string.

OUTPUT PARAMETERS:  NONE

MODULES CALLED:  (The symbol * designates a primitive)
*shall
*writeln

EXIT CONDITIONS:
Normal exit conditions.

REMARKS:  NONE
APPENDIX B

PASCAL CODE OF ROUTINES
OF THE PROTOTYPE FOR TS
SPECIFICATIONS
(User A's side only)
NAME: BUFFCLEAR

FUNCTION: To clear the event flag corresponding to the passed integer value.

USAGE: procedure buffclear(eFn : eFnn);

buffclear(eFnn);

INPUT PARAMETERS:  eFnn  the event flag number.

OUTPUT PARAMETERS: NONE

MODULES CALLED:

EXIT CONDITIONS:
If the return code from the CLEF$ Directive is less
than zero then an error message is printed and
the event flag may not have been cleared and
the processor is halted using the 'halt' primitive.

REMARKS:
See RSX-11M V3.2 Executive Directives Manual on page 6-26 for
more information.

(type  
  eFn  = integer

procedure buffclear(eFn : eFnn);

var  ids : integer;

procedure cIref(var  eIn : integer; var  ids : integer); extern(fortran);)
BUFFCLEAR.PAS

BUFFCLEAR

44 begin
45 writeln('Entering buffclean routine');
46 clref(esr, ids); (* Call CLEF *)
47 if ids < 0 then (* Check if return code is less than 0 *)
48 begin
49 writeln('Error in call to clref. Error code - ids');
50 halt;
51 end;
52 end;
53 (writeln('Leaving buffclean routine'));
54 end.

NO ERROR DETECTED
TOTAL PROGRAM SIZE 00025A
OUTERMOST DATA SIZE 000002
RESERVED STACK & HEAP 000222
COMPILATION TIME 8 SECONDS

BUFFCLEAR.BUFFCLEAR/P42-BUFFCLEAR
NAME: BUFFGO

FUNCTION: To set the event flag corresponding to the passed integer value.

USAGE: procedure buffgo(efn : efn);

buffgo(efn);

INPUT PARAMETERS: efn - the event flag number.

OUTPUT PARAMETERS: NONE.

MODULES CALLED:

>halt
>writeln
>self

EXIT CONDITIONS: Normal exit conditions.

Also see chapter 3 of the manual for more information on event flags and their use.

REMARKS:

See RSX-11M V3.2 Executive Directives Manual on page 6-123 for more information.

Also see chapter 3 of the manual for more information on event flags and their use.

***********************************************************************************************

type

  efno = integer;

procedure buffgo(efn : efno);
var e, ids : integer;

44  procedure setef(var efn:integer; var ids : integer); writeln('entering buffer routine');
46  begin
48  writeln('entering buffer routine');
49  setef(efn,(ids):--(# call the SETF$ directive #)
51  if ids < 0 then
52  begin
53  writeln('error in setef. Error code: ', ids);
54  halt
55  end
56  end
57  writeln('leaving buffer routine');
58  end.

NO ERROR DETECTED
TOTAL PROGRAM SIZE 000250
EXTERNAL DATA SIZE 000002
RESERVED STACK & HEAP 000022
COMPILATION TIME 6 SECONDS

PAS BUFFGO;BUFFGO/P42-BUFFGO
(****a-*)
(***************************************************************************

NAME: BUFWAIT

FUNCTION: To wait until the event flag corresponding to the passed integer value

USAGE: procedure bufwait(event: integer);

bufwait(event);

INPUT PARAMETERS: event - the event flag number.

OUTPUT PARAMETERS: NONE

MODULES CALLED: halt

write

EXIT CONDITIONS: Normal exit conditions.

if the return code from the WTECH Directive is less
than zero then an error message is printed and
the processor is halted using the 'halt' directive.

REMARKS: See RSX-11M V3.2 Executive Directives Manual on page 6-166 for more
information.

One should be aware of the different types of event flags that
can be used for different purposes since for example if global
event flags are used for controlling events in a particular task
then one must make sure that these flags are not used somewhere
else in the system (see chapter 3 of manual).

***************************************************************************

type event = integer;
procedure buffwait(var fn: string);

var ids : integer;

procedure waitr(var fn: string; var ids: integer); external;

begin
  writeln('Entering buffwait routine');
  waitr(fn, ids);  (* wait for file to be set *)
  if ids = 0 then  (* was it successful? *)
    begin (* no *)
      writeln('error in waitr. Error code = ', ids);
      halt;
    end;
  writeln('Leaving buffwait routine');
end.

NO ERROR DETECTED
TOTAL PROGRAM SIZE  000252
OUTERMOST DATA SIZE  000002
RESERVED STACK & HEAP  000222
COMPILATION TIME 7 SECONDS

FAS BUFFWAIT,BUFFWAIT/P42-BUFFWAIT
NAME: READERM

FUNCTION: To read in a string from standard input and record its length.

USAGE: procedure request(var buffer : request ; var len : integer) external

readerem(buffer : len);

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: buffer - the string buffer.
len - the length of the string buffer.

MODULES CALLED:
break
deolin
read
readln
write

EXIT CONDITIONS: Normal condition.

REMARKS:
The input line is truncated to REQLEN; care should be taken not to exceed this value.

It is not recommended to input non-printable characters using this module since unpredictable behavior can result when these strings are output.

const REQLEN : 61; (GLOBAL CONSTANT)
type request = array [1..REQLEN] of char;
procedure readerem(var buffer : request ; var len : integer);
const
  revidlo: '7m';
  video: '0m';
  up: 'A';

var
  i: integer;

begin
  writeln(CHR(27) + revidlo + CHR(27) + up);
  (set input into reverse video $)
  writeln(CHR(27) + revidlo + CHR(27) + up);
  (set input into reverse video $)
  break(tty):;
  readln(tty):;
  i := 0;
  if not(eolnttys) then
    begin
      i := i;
      (length of string must not exceed RELEN $)
    end
    while (not (eolnttys) and (i <= RELEN)) do
      begin
        read(tty, buffer(i));
        i := i + 1
        end
        len := i - 1;
  end
  else
  writeln(CHR(27) + video + CHR(27) + up); 
  (set back in normal display mode $)
  end.

NO ERROR DETECTED
TOTAL PROGRAM SIZE   .000840
EXTERNAL DATA SIZE    .000002
RESERVED STACK & HFAP .000522
COMPILED TIME 11 SECONDS

PAS READTERM;READTERM;F42-READTERM
**NAME:** RECA

**FUNCTION:** To receive some information sent to the TRNSDA task by using the receive data RECEIUV directive and to return the status of the call.

**USAGE:**

```
procedure receiv(var ibuf : ibufi,
                 var ids : ids); external
```

**INPUT PARAMETERS**

NONE

**OUTPUT PARAMETERS:**

ibuf - the returned buffer
ids - the returned status of the directive call.

**MODULES CALLED:**

(The symbol '*-' designates a primitive)

```
'ord
'writeln
receiv
```

**EXIT CONDITIONS:**

The return code of RECEIUV is set to greater or equal to 0 (zero) in case of successful completion and to less than 0 in case of unsuccessful completion.

For more information on the use of this directive and its return codes, see the RSX-11M V3.2 Executive Reference Manual.

**REMARKS:** NONE

```
const
```

__________________________________________________________
RECA.PAS

MAIN.

44 REQUESTLEN = 61
45 BUFLEN = 131
46 DUMLEN = 81

48 typ
49 transDxresk = (w, x, y, z, i, n, e, p);
50 request = packed array[1..REQUESTLEN] of char
51
tskk = record
52    nm : array[1..2] of integer
53 end;
54
55 rbuff = record
56    nm : tskk;
57    req : request;
58    by : boolean;
59    ch : transDxresk;
60    dnm : array[1..DUMLEN] of integer;
61 end;
62
63 efinn = integer;
64
65 idss = integer;
66
67 procedure rece(var rbuff : rbuff);
68 var ids : idss);
69
70 var
71    dummy : efinn;
72      tsk : tskk;
73
74 procedure receiv(var tsk : tskk;
75 var rbuff : rbuff;
76 var dummy : efinn;
77 var ids : idss)) extern(fortran);
78
79 begin
80    writeln('entering rece procedure');
81    receiv(tsk, rbuff, dummy, ids);
82    receiv(tsk, rbuff, dummy, ids);
83    receiv(tsk, rbuff, dummy, ids);
84
din
FASCAL PDF-11 VERSION 8.07  1983-08-06  12:35:33  PAGE 3
RECA.PAS

87  writeln('user request = ' + rbstr + ' use char = ' + ord(rbstr))
88
89  writeln('leaving reca procedure')
90  end.

NO ERROR DETECTED
TOTAL PROGRAM SIZE  000106
OUTFRHOST DATA SIZE  000002
RESERVED STACK & HEAP  000526
COMPILATION TIME 6 SECONDS

IA:  RECA/RECA/P42-RECA
NAME: TERMA (MAIN)

FUNCTION: This module is used to continuously READ information from USER A (TERMINAL A) and to send this information to the TRNSDA task. This information is obtained from terminal input in the form of user primitive strings.

USAGE: from ACR: 'RUN TERMA /TASK-TERMA'.

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: NONE

MODULES CALLED: (The symbol '.', designates a primitive)
- halt
- read
- writeln
- buffclear
- boffdo
- buffwait
- readterm
- send

EXIT CONDITIONS: This program has been designed to run indefinitely until the task into which it runs is specifically aborted with the ACR command "ABORT TERMA".

If any errors are detected by one of the routines called, then the routine itself will print an error message and halt the processor. The exception to this is the call to the STAT ("send") routine. In this case TERMA checks the return status and in case of error prints out an error message and then halts the processor by using the 'halt' primitive.

REMARKS: NONE
program termally;

const
TERMFLAG = -421;
TRANSDAFLAG = 431;
REQUESTLEN = 61;
BUFLEN = 151;
DUMLEN = 81; (* BUFLEN - 2 - (REQUESTLEN + 1) / 2 *)

type
  transXresp = (var or 3; driver :); (*
request = packed array[1..REQUESTLEN] of char;*
  buff = record
    req : request
  typ : boolean;
    ch : transXresp;
    dum : array[1..DUMLEN] of integer;
  end;
  tsk  = record
    num : array[1..2] of integer;
  end;
  efn  = integer;
  idss = integer;

var
  buf : buff;
  efn : efn;
  ids : idss;
  j : integer;
  len : integer;
  tsk : tsk;

procedure buffclear(efn : efn); external
procedure buffset(efn : efn); external
procedure buffwait(efn : efn); external
procedure readterm(var req : request;
                  var len : integer); extern

procedure send(var tsk : tsk;
               var buf : buf);

begin
  write('Task TERNA activated
        
(*) TERNA *)
  writeln('Task TERNA activated.........');
  (* set up the receiver task name 'TRNSDA'*)
  tsk.name[1] := 32734;
  tsk.name[2] := 30562;
  (* set up event flag to wake up receiver task *)
  efn := TRNSDAFLAG;
  (* set up type of transfer (string) *)
  buf.type := true;
  (* initialize the rest of the buffer to be sent *)
  buf.ch := a;
  buf.dom[3] := 31;
  buf.dom[5] := 51;
  buf.dom[7] := 71;
  buf.dom[8] := 81;
  while (1 = 1) do
    begin
      (* put user primitive into the appropriate buffer field *)
      readterm(buf, req, len);
      (* pad string with blanks if needed *)
      for j := (len + 1) to REQUESTLEN do
        buf.req[j] := ' ';
      (* wait until TERNA has priority to send to TRNSDA *)
buffwait(TERMAFLAG);

(* send data buffer to TRNSDA *)

send(buff, nfin, lds);

if (id < 0) then

begin

writeln(tty, 'error in calling SEND in TERMA, error code - ', id);

halt

end

end;

writeln(tty, 'Task TERMA terminated...........

NO ERROR DETECTED
TOTAL PROGRAM SIZE 001534
OUTERMOST DATA SIZE 000050
RESERVED STACK & HEAP 001100
COMPILATION TIME 15 SECONDS

PAS TERMA,TERNA/F42-TERMA
NAME: TRNSDA (MAIN)
FUNCTION: Transducer for side 'A'.
USAGE: from MCR: 'RUN TRNSDA/TASK-TRNSDA'

INPUT PARAMETERS: NONE
OUTPUT PARAMETERS: NONE

MODULES CALLED: (The symbol '*' designates a primitive)

>halt
>write
>written
>buffclear
>buff鼓舞
>buffwait
>mark
>reca
>waab
>utter

EXIT CONDITIONS: This program has been designed to run indefinitely until
the link into which it runs is specifically aborted with
the MCR command 'ABORT TRNSDA'.

If any errors are detected by one of the routines called,
then the routine itself will print an error message and
halt the processor by using the 'halt' primitive. The
exception to this is the RECA routine. TRNSDA checks
the status of the call to RECA and if there is an error
condition then an error message is printed and the
processor is halted by TRNSDA.

REMARKS: See Ref. (22) for more details.

******************************************************************
program trn3da(lly);

const
TERMFLAG = 40;
TRNSDBFLAG = 41;
TERMAFLAG = 42;
TRNSDAFLAG = 43;
REQUESTLEN = 64;
BUFFLEN = 131;
DUMLEN = 81 + (BUFFLEN - 2 - (REQUESTLEN + 1) / 2);

type
transXresp = (ok, err, or, in, ex);
request = packed array[1..REQUESTLEN] of char;

lskk = record	nam : array[1..2] of integer;
end;

rbuff = record
  nam : lskk;
  req : request;
  typ : booleani;
  ch : transXresp;
  dum : array[1..DUMLEN] of integer;
end;

buff = record
  req : request;
  typ : booleani;
  ch : transXresp;
  dum : array[1..DUMLEN] of integer;
end;

state = (one, two, three, four);
efnm = integer;
ldss = integer;
tntt = integer;
```pascal
var
  b : char;
  buf : buffer;
  efn : efun;
  ids : idax;
  rbuf : rbuffer;
  state : state;
  tag : tag;
  tnt : tunt;

procedure buffClear(efn : efun); external;
procedure buffAdd(efn : efun); external;
procedure buffWait(efn : efun); external;
procedure mark(var efn : efun);
var
  tag : tag;
  tnt : tunt;
  ids : idax; extern(fortran);

procedure recall(var rbuf : rbuffer;
var ids : idax); external;
procedure waal(messagex : transxres); external;
procedure wterma(messagex : integer;
state : state, 
ddd : request); external;

begin (* TRNSDA *)
  (* initialize state to IDLE state *)
  state : - one;
writeln('TRNSDA is activated............');
  (* initialize waiting in between empty queues *)
  tag := 2; (* 2 *)
  tnt := 2; (* seconds *)
```
while (1 - 1) do
begin
begin
  (* wait until TRNSDA has priority *)
  buffwait(TRNSDAPFLAG);
  (* remove priority from TERMA *)
  buffclear(TERMAFLAG);
  (* receive data from TRNSDA task queue *)
  rec(rlbuf, ids);
  if (ids = 1) then
  begin
    if rlbuf.lw then (* message from terminal A *)
    begin
      case stateXa of
        (* select the proper state *)
        (* if in idle state then do the following *)
        one:
        begin
          if (rlbuf.rea = 'textarea') then
          begin
            stateXa := two
            (* outXconnXpending *)
            whsh(d)
          end
          else
          begin
            whs(rlbuf) (* idle *)
            (* if in outXconnXpending state then do the following *)
          end
          two:
          begin
            if (rlbuf.rea = 'textarea') then
            begin
              stateXa := one
              (* back from idle state *)
              whsh(d) (* SEND data to TRNSDB *)
            end
            else
three: begin
  if (rbuf.rea - 'hreu') then
    begin
      stateXa := one;
      wuab(d);
    end
  else if (rbuf.rea - 'lores') then
    begin
      stateXa := four;
      wuab(s);
    end
  else
    wterm(s, stateXa, rbuf.rea)
end; (* INXCONNXPENDING *)

four: begin
  if (rbuf.rea - 'lreu') then
    begin
      wuab(d);
      stateXa := one;
    end
  else if (rbuf.rea - 'hndr') then
    begin
      wuab(s);
    end
  else if (rbuf.rea - 'lexdr') then
    begin
      wuab(s);
    end
  else
    wterm(s, stateXa, rbuf.rea)
end; (* dataXtransferXrdy *)
216        end
217       end
218   end
219
220   (* the following section deals with the messages coming in from the
221      uueux0xu0xa
222   *)
223   else if (rbuf.ch = u) then
224     wterma(7; stateXa; rbuf, rea)
225   else case stateXa of
226       (* if in idle state then do the following *)
227       one: begin
228         if (rbuf.ch = u) then
229           begin
230             wterma(2; stateXa; rbuf, rea);
231             stateXa := three
232           end
233         else
234           wuab(e)
235         end
236       end
237       (* idle *)
238
239     (* if in outXcomXpending state then do the following *)
240   two: begin
241     if (rbuf.ch = d) then
242       begin
243         wterma(3; stateXa; rbuf, rea);
244         stateXa := one
245       end
246     else if (rbuf.ch = s) then
247       begin
248         wterma(1; stateXa; rbuf, rea);
249         stateXa := four
250       end
251   end
252   else
begin
  waab(e)
end;
end;

(* SEND data to NTRNSDB *)

(* OUTXCONXXPENDING *)

(* if in inXcomXXpending state, then do the following *)

three:
begin
  if (rbuf.ch = d) then
  begin
    wterma(3, stateXa, rbuf, reu);
    stateXa := one;
  end
  else
  begin
    waab(e)
  end;
end;

(* inXcomXXpending *)

(* if in dataTransferXrdy state, then do the following *)

four:
begin
  if (rbuf.ch = d) then
  begin
    wterma(3, stateXa, rbuf, reu);
    stateXa := one;
  end
  else if (rbuf.ch = i) then
  begin
    wterma(4, stateXa, rbuf, rea)
  end
  else if (rbuf.ch = x) then
  begin
    wterma(5, stateXa, rbuf, reu)
  end
  else
  begin
    waab(e)
  end;
end;

(* dataTransferXrdy *)
end

end

else if (ids <= -8) then

begin

writeln(tty, 'error in reading from TRNSDA task queue');

halt

end;

/* give priority to other terminal */

buffgo(TERMFLAG);

(* start up TRNSDB after 2 seconds *)

efn := TRNSDBFLAG;

mark(efn tag, lntr ids);

(* reset priority of TRNSDA *)

buffclear(TRNSDAFLAG);

end;

(* end of RUN forever *)

writeln(tty, 'TRNSDA TASK TERMINATED....................');

end.

NO ERROR DETECTED
TOTAL PROGRAM SIZE  004530
OUTERHOST DATA SIZE  000106
RESERVED STACK & HEAP  001134
COMPILATION TIME 34 SECONDS

PAS TRNSDA:TRNSDA/PAS-TRNSDA
NAME: WQAB

FUNCTION: To put a symbol into the A-R queue.

USAGE: procedure WQAB(messageXu: transX; extX)

waab(messageXu);

INPUT PARAMETERS: messageXu - the symbol to be sent to the queue.

OUTPUT PARAMETERS: NONE

MODULES CALLED: (The symbol '*' designates a primitive)

halt
writeln
send

'EXIT CONDITIONS:

The return code of SDAT is set to greater or equal to 0 (zero) in case of successful completion and less than 0 in case of unsuccessful completion.

For more information on the use of these system routines and their return codes, see the RSX-11M V3.2 Executive Reference Manual.

REMKS: NONE

const QABFLAG = 361
REQUESTLEN = 6;
BUFLEN = 13;
DUMLEN = .8; (* BUFLEN - 2 - (REQUESTLEN + 1) / 2 *)

type
transdXresr = (a, ar, br, cr, dr, er, fr, gr, hr, ir, jr, kr, lr, mr, nr, or, pr, qr, rr, sr, tr, ur, vr, wr, xr, yr, zr, z);
request = packed array[1..REQUESTLEN] of char

buff = record
  req : request;
  typ : boolean;
  ch : transdXresr;
  dum : array[1..DUMLEN] of integer;
end;

task = record
  name : array[1..2] of integer;
end;

efnn = integer;
idss = integer;

procedure waab(messagex : transdXresr);
var
  buf : buffi;
  efn, idss, len : integer;
  task : taski;
begin
  writeln(task, ' entering waab procedure');
  (* set up task name = 'TRANSDB' *)
87   tsk.nam[1] := 32734;
88   tsk.nam[23] := 30562;
89   
90   (* set up event flag to wake up TRNSDB *)
91   ef1 := QAFLAG;
92   
93   (* set boolean field to queue type *)
94   buf.type := false;
95   
96   (* put symbol into the buffer field of the send buffer *)
97   buf.ch := 'message'
98   
99   (* send the data to the TRNSDB task *)
100  send(tsk, buf, ef1, ids);
101  if ids < 0 then
102    begin
103      writeln(tty, 'error in calling SEND in WQAB, error code = ', ids);
104      halt
105    end;
106   
107   (* writeln(tty, 'leaving wqab procedure') *)
108  end.

NO ERROR DETECTED
TOTAL PROGRAM SIZE    000470
OUTERMOST DATA SIZE   000002
RESERVED STACK & HEAP 000064
COMPILATION TIME 9 SECONDS

PAS WQAB,WQAR/P42-WQAB
NAME: WTERNA

FUNCTION: To write a TRNSDA error message on the terminal in case illegal requests are made by TRNSDB or terminal A or to write a query status message or one of the following messages (successful): become blind, blind, blind.

USAGE: procedure wterm(messno: integer;
state: integer;
req: request); external;

INPUT PARAMETERS: messno - the type of message;
state - the state of TRNSDA;
req - the input string.

OUTPUT PARAMETERS: NONE

MODULES CALLED: (The symbol . designates a primitive)
hal -
written

EXIT CONDITIONS: Normal exit conditions.

REMARKS: NONE

const REQUESTLEN = 61;

type:
request = packed array[1..REQUESTLEN] of char;
state = (one, two, three, four);

procedure wterm(messno: integer; istate: state; ireq: request);
const
{ LOCAL VI100 TERMINAL CONSTANTS }
re video = '7';
video = '0';
up = '9';

begin

(* put in reverse video display mode *)
write(v, chr(27), re video, chr(27), up);

if rea = 9 then

begin
write(tty, 'NTRNSDA is in ');
case state of
one : write(tty, 'IDLE ');
two : write(tty, 'OUT CONNECT PENDING ');
three: write(tty, 'IN CONNECT PENDING ');
four : write(tty, 'DATA TRANSFER READY ');
end;
writeln(tty, 'state. ');
end

else case messno of
1  : writeln (TTY, 'cc');
2  : writeln (TTY, 'led');
3  : writeln (TTY, 'led');
4  : writeln (TTY, 'led');
5  : writeln (TTY, 'led');
6  : begin
write(tty, 'NTRNSDA cannot accept a ', rea, ' in ');
case state of
one : write(tty, 'IDLE ');
two : write(tty, 'OUT CONNECT PENDING ');
three: write(tty, 'IN CONNECT PENDING ');
four : write(tty, 'DATA TRANSFER READY ');
end;
writeln(tty, 'state. ');
end

end
(* case *)
(* put back in normal mode *)
87 writeln(chr(27) + " videor chr(27) + ur\r\n"
88 end.

NO ERROR DETECTED
TOTAL PROGRAM SIZE 002336
OUTERHOST DATA SIZE 000002
RESERVED STACK & HEAP 000520
COMPILED TIME 17 SECONDS

PAS WTERMA; WTERMA/F42=WTERMA
APPENDIX C

DESCRIPTION OF ROUTINES
OF THE PROTOTYPE FOR
CLASS-0 TP SPECIFICATIONS
(User A's side only)
NAME: BUFFCLEAR

FUNCTION: To clear the event flag corresponding to the passed integer value.

USAGE: procedure buffclear(efn : efn);

buffclear(efn);

INPUT PARAMETERS: efn - the event flag number.

OUTPUT PARAMETERS: NONE

MODULES CALLED: >halt
                >writeln
                cref

EXIT CONDITIONS:
If the recode from the CLEF* Directive is less
than zero then an error message is printed and
the event flag may not have been cleared and
the processor is halted using the 'halt' primitive.

REMARKS: See RSX-11M V3.2 Executive Directives Manual on page 6-26 for
more information.
NAME: BUFFGO

FUNCTION: To set the event flag corresponding to the passed integer value.

USAGE: procedure buffgo(efn: efnn);

buffgo(efn);

INPUT PARAMETERS: efn — the event flag number.

OUTPUT PARAMETERS: NONE

MODULES CALLED:
>halt
>writeln
setef

EXIT CONDITIONS: Normal exit conditions.
if the retcode from the SETEF% Directive is less
than zero then an error message is printed and the
event flag may not be set and the processor is halted
using the 'halt' primitive.

REMARKS:
See RSX-11M V3.2 Executive Directives Manual on page 6-123 for
more information.

Also see chapter 3 of the Manual for more info on event flags and
their use.
NAME: BUFFWAIT

FUNCTION: To wait until the event flag corresponding to the passed integer value is set.

USAGE: procedure buffwait(efn : efnn);

buffwait(efn);

INPUT PARAMETERS: efn - the event flag number.

OUTPUT PARAMETERS: NONE

MODULES CALLED: >halt
> writeln
waitfr

EXIT CONDITIONS: Normal exit conditions.

if the recode from the WTE$ Directive is less than zero then an error message is printed and the processor is halted using the 'halt' primitive.


One should be aware of the different types of event flags that can be used for different purposes since for example if global event flags are used for controlling events in a particular task then one must make sure that these flags are not used somewhere else in the system (see Chapter 3 of Manual).
NAME: FILLB

FUNCTION: This procedure converts an integer to character format into the
given character field.

USAGE:

```
procedure fillb(b: integer;
    var c: char); external;

fillb(b, c);
```

INPUT PARAMETERS: $b$ - the integer to convert.

OUTPUT PARAMETERS: $c$ - the character field to be assigned.

MODULES CALLED:
(The symbol '=>' designates a primitive)

```
=>chr
=>halt
=>writeln
```

EXIT CONDITIONS: If the passed integer is $> 256$ or $< 0$ then
an error message is printed on the terminal
and the processor is halted.

REMARKS: For Modularity and Portability reasons the primitive
'chr' should have been replaced by fillb in all the
modules in this system. This should definitely be
considered for future enhancements.
NAME: FILLW

FUNCTION: This procedure converts the passed integer to its equivalent two characters LO and HI.

USAGE: procedure fillw( b : integer;
var c1 : char;
var c2 : char); external;

fillw(b, c1, c2);

INPUT PARAMETERS: b - the integer to convert.

OUTPUT PARAMETERS: c1 - the LO field of the integer.
c2 - the HI field of the integer.

MODULES CALLED: (The symbol '*' designates a primitive)

>halt
>writeLn

EXIT CONDITIONS: Normal exit conditions

REMARKS: It is to note here that the PDP-11 stores an integer's hi and low bytes in inverse order in memory. That is LO-HI.
**NAME:** INITIAL

**FUNCTION:** This module is used to create a dynamic region for a task using the CRG routine by issuing a CRG$ memory management directive.

**USAGE:**

```pascal
procedure initial(var id : integer); external;

initial(id);
```

**INPUT PARAMETERS:** NONE

**OUTPUT PARAMETERS:** id - the region id.

**MODULES CALLED:** (The symbol $^p$ designates a primitive)

```pascal
>halt
>writeln
>crdg
```

**EXIT CONDITIONS:**
If there is an error detected in calling CRG$, then an error message identifying the error and the error code is printed onto standard I/O and the processor is then halted by using the "halt" primitive.

The return code of the CRG$ routine is set to greater or equal to 0 (zero) in case of successful completion and to less than 0 in case of unsuccessful completion.

For more information on the use of CRG$ and of its return codes, see pages 6-36 to 6-38 of the RSX-11M V3.2 Executive Reference Manual.

**REMARKS:** At the moment, there is no protection on the region. One can easily remedy this by giving an appropriate value to the region protection word.
NAME: INITBUF

FUNCTION: To initialize a TPDU with zeros in all of its fields.

USAGE: procedure initbuf(var swndno : std); external;

initbuf(swndno);

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: swndno - the record to be filled with zeros in all of its fields (initialization).

MODULES CALLED: ( '\' indicates a primitive )
>chr
>writeln

EXIT CONDITIONS: Normal condition.

REMARKS: The FILLB routine should probably have been used here instead of chr.
(*********************************************************)

NAME: INSPECTDU

FUNCTION: To inspect a TPDU for erroneous parameter values and returns the length of the erroneous packet if an error is detected and sets valid to FALSE; otherwise sets valid to TRUE.

USAGE: procedure inspectdu(ptrnd, valid, rjcol, err);

   var valid : boolean;
   var rjcol : integer;
   var err : error; extern

inspectdu(ptrnd, valid, rjcol, err);

INPUT PARAMETERS: ptrnd - the pointer to the TPDU to inspect.

OUTPUT PARAMETERS: Valid - set to FALSE if error is detected and to TRUE otherwise.

rjcol - the detected cause for rejection of TPDU.

err - the length of the erroneous string which is sent back.

MODULES CALLED: (The symbol '*' designates a primitive)

 ord
 writeln
 val

EXIT CONDITIONS: If an error is detected in the received buffer,
the length of the erroneous string is returned with valid = FALSE
otherwise valid is set to TRUE.

REMARKS: Length returned by this routine is up to and including the first erroneous parameter or STRLEN
in case of successful inspection.

At present, the inspection performed is not very sophisticated since it is basically done by comparing the different fields of a particular TPDU with some fixed global constants which represent the only value acceptable. For our purposes this method has proven to be sufficient.

))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
NAME: NTAINIT

FUNCTION: This module is used to create a dynamic region for
task NTAINIT by issuing a CRRG$ memory management directive.

USAGE:

```
procedure ntainit(var id : integer); external
```

```
ntainit(id);
```

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: id - the region id.

MODULES CALLED:

```
&halt
&writeln
```
crrg

EXIT CONDITIONS:

If there is an error detected in calling CRRG$, then
an error message identifying the error and the error
code is printed onto standard I/O and the processor
is then halted by using the "halt" primitive.

The return code of the CRRG$ routine is set to greater
or equal to 0 (zero) in case of successful completion
and to less than 0 in case of unsuccessful completion.

For more information on the use of CRRG$ and its
return codes, see pages 6-36, 6-38 of the RSX-11H V3.2

REMARKS: At the moment, there is no protection on the region.
One can easily remedy this by giving an appropriate
value to the region protection word.
NAME: NTAREC

FUNCTION: To wait until some information is sent to the NTRNSDA task via send data or send by reference and receives it via receive data or receive by reference.

USAGE: procedure ntarec(var ptrwnd : psdl;
    var nXreu : netXrea;
    var queue : boolean;
    var messXtrenta : boolean); external;

    ntarec(ptrwnd, nXreu, queue, messXtrenta);

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS:
    ptrwnd - the pointer to receive by reference buffer.
    nXreu - receive data string or receive by reference string.
    queue - true if a receive data was performed
     false if a receive by reference was done.
    messXtrenta - true if data was sent by TPRENTA task
     false if data was sent by NTRNSDB task.

MODULES CALLED: (The symbol '>' designates a primitive)
    >chr
    >halt
    >writeln
    buffwait
    crw
    ulr-
    märk
    receiv
    rref

EXIT CONDITIONS: If there are any errors detected in calling CRAW$, DTRG$.
MARK$, RECEIVE$ or RREF$, then an error message identifying
the error and error code is printed onto standard I/O and
the processor is then halted by using the 'halt' primitive.

The return code of all these system directives is set to
greater or equal to 0 (zero) in case of successful completion
and to less than 0 in case of unsuccessful completion.

For more information on the use of these system routines
and their return codes, see the RSX-11M V3.2 Executive

REMARKS:
When there is a receive by reference, the newly attached region
is detached in order to prevent depletion of the memory pool.
NAME: NTRNSDA (MAIN)

FUNCTION: Network transducer for side 'A'.

USAGE: from HCR: "RUN NTRNSDA /TASK-TRNSDA /INC=100."

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: NONE

MODULES CALLED: (The symbol '>' designates a primitive)

>chr
>new
>ord
>write
>writeln
>buffer
>bufso
>buffwait
>ntainit
>ntarec
>sdtrenta
>waab
>waess

EXIT CONDITIONS: This program has been designed to run indefinitely until
the task into which it runs is specifically aborted with
the HCR command 'ABORT=TRNSDA'.

If any errors are detected by one of the routines called,
then the routine itself will print an error message and
halt the processor by using the 'halt' primitive.

REMARKS: See Ref. (22) for more details.
NAME: PRETPDU

FUNCTION: To prepare a TPDU according to the TPDU code.

USAGE: procedure prepdu (strnd : ptdt;
   var dstref : enc;
   var srcref : enc;
   var ruse : integer;
   var err : enc;
   var cod : integer);

prepdu(strnd, dstref, srcref, ruse, err, cod);

INPUT PARAMETERS:
   strnd - the pointer to the packet to be filled.
   dstref - destination address.
   srcref - source address.
   ruse - cause for rejection of TPDU.
   err - length of the erroneous TPDU string.
   cod - the type of TPDU packet.

OUTPUT PARAMETERS:
   strnd - the pointer to the TPDU.
   dstref - destination address.
   srcref - source address.

MODELS CALLED: (The symbol "\*" designates a primitive)

   >chr
   >writeln

EXIT CONDITIONS:
   Normal exit conditions.

REMARKS:
   Presently, most of the filling is done in TERTMA. For future enhancements, the filling should be done by this module.

   See the TREPORT module for more information about the format of the different TPDUs.
NAME: READUDATA

FUNCTION: To read in a string from standard input into the appropriate field of a particular packet, record its length and put it into the correct length field of this packet.

USAGE: procedure readUdata(var w : std); external

readUdata(w);

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: w - the packet to be filled with a string and its associated length.

MODULES CALLED:
>break
>eoln
>read
>readln
>writeln
>fillb

EXIT CONDITIONS: Normal condition.

REMARKS:
The input line is truncated to STRLEN, so care should be taken not to exceed this value.
The user (TERMINAL A) should make sure that his/her terminal buffer is at least \geq STRLEN. This can be checked by issuing the MCR command \texttt{SET /BUF-TI}. If the buffer is too small, then this can be altered by the MCR command \texttt{SET /BUF-TI:200.} if a buffer of 200 bytes is desired for example.

Unpredictable behavior results when non-printable characters, such as escape or end-of-file characters, are input by this module.
**NAME:** READTERM

**FUNCTION:** To read in a string from standard input and record its length.

**USAGE:**
```pascal
procedure request(var buffer : request ; var len : integer) external;
readterm(buffer, len);
```

**INPUT PARAMETERS:** NONE

**OUTPUT PARAMETERS:**
- buffer — the string buffer.
- len — the length of the string buffer.

**MODULES CALLED:**
- break
- eoln
- read
- readln
- writeln

**EXIT CONDITIONS:** Normal condition.

**REMARKS:**
- The input line is truncated to REQLEN, so care should be taken not to exceed this value.
- It is not recommended to input non-printable characters using this module since unpredictable behavior can result when these strings are outputted.
NAME: RECCA

FUNCTION: To wait until some information is sent to the IPRENTA Task via send data or send by reference, and receives it via receive data or receive by reference.

USAGE:

```pascal
procedure recca(var pString : pstdi;
    var nReu : intReu;
    var queue : boolean;
    var messXterm : boolean); extern

recca(pString, nReu, queue, messXterm);
```

INPUT PARAMETERS

None

OUTPUT PARAMETERS:

- `pString` - Pointer to received by reference buffer.
- `nReu` - Receive data or reference string.
- `queue` - True if a receive data was performed.
- `messXterm` - True if data was sent by TERMA task.

MODULES CALLED:

(The symbol `*` designates a primitive)

- `>chr`
- `>halt`
- `>writeln`
- `buffwait`
- `craw`
- `dtfs`
- `mark`
- `receiv`
- `reff`

EXIT CONDITIONS:

If there are any errors detected in calling CRAW$, DTGS$, MARK$, RECEIV$ or RREF$, then an error message identifying
the error and error code is printed onto standard I/O and the processor is then halted by using the 'halt' primitive.

The return code of all these system directives is set to greater or equal to 0(zero) in case of successful completion and to less than 0 in case of unsuccessful completion.

For more information on the use of these system routines and their return codes, see the RSX-11M V3.2 Executive Reference Manual.

REMARKS: When there is a receive by reference, the newly attached region is detached in order to prevent depletion of the memory pool.

NAME: SDTPRENTA

FUNCTION: To send information to task TFRENTA via system directives:
send data (SDAT$) or send by reference (SREF$).

USAGE: procedure sdtprenta(channel : ref;
var wndno : wndno;
var xrea : metxrea;
var id : integer;
var bol : boolean); extern

sdtprenta(channel, wndno, xrea, id, bol);

INPUT PARAMETERS:
channel - pointer to the buffer to be sent.
wndno - the window number to be sent if bol = false.
xrea - the string to be sent.
id - the region id.
bol - the boolean switch.

OUTPUT PARAMETERS:
wndno - the next window number.

MODULES CALLED:
(The symbol '?' designates a primitive)

>halt
>writeln
craw
sref
sdat

EXIT CONDITIONS:
If there are any errors detected in calling CRAW$, SDAT$,
or SREF$, then an error message identifying the error and
error code is printed onto standard I/O and the processor
is then halted by using the 'halt' primitive.

The return code of all these system directives is set or
greater or equal to 0 (zero) in case of successful completion.
and less than 0 in case of unsuccessful completion.

For more information on the use of these system routines and their return codes, see the RSX-11M V3.2 Executive Reference Manual.

REMARKS:

See TSEND routine for information on method of circular window queues and its restrictions.
NAME:       SENDD

FUNCTION:   To send information to task NTRNSDA via system directives
            send data (SDAT$) or send by reference (SREF$).

USAGE:      procedure sendd(ptrwnd, pstdi
            var wndno: wndno;
            nxrea, ntxreq: integer;
            id: integer;
            bol: boolean); extern

            sendd(ptrwnd, wndno, nxrea, id, bol);)

INPUT PARAMETERS:
            ptrwnd - the pointer to the buffer to be sent.
            wndno  - the window number to be used.
            nxrea  - the string to be sent if bol = true.
            id     - the region id.
            bol    - the boolean switch.

OUTPUT PARAMETERS:  wndno  - the next window number.

MODULES CALLED:
            (The symbol '*' designates a primitive)

            '>halt
            '>writeln
            crw
            sref
            sdat

EXIT CONDITIONS:
            If there are any errors detected in calling CRAW$, SDAT$ or SREF$, then an error message identifying the error and
            error code is printed onto standard I/O and the processor
            is then halted by using the 'halt' primitive.

            The return code of all these system directives is set or
            greater or equal to 0 (zero) in case of successful completion
            and less than 0 in case of unsuccessful completion.
For more information on the use of these system routines and their return codes, see the RSX-I1M V3.2 Executive Reference Manual.
NAME: TERMA (MAIN)

FUNCTION: This module is used to continuously READ information from
USER A (TERMINAL A) and to send this information to the TPRENTA
task. This information is obtained from terminal input, in the
form of user primitive strings and also data strings (see tnmbr
and treno primitives), and also from certain text files which
contain information related to the selected user primitive.

USAGE: from HCR : "RUN TERMA /TASK=TERMA /INC-100."

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: NONE

MODULES CALLED: (The symbol '*' designates a primitive)

>chr'
>init
>ioreresult
>read
>readin
>reset
>writeln
>bufc1ear
>buffdo
>buffwait
>fillb
>fillw
>initbuf
>readera
>readdata
>limit
>tsend

EXIT CONDITIONS: This program has been designed to run indefinitely until
the task into which it runs is specifically aborted with
the HCR command "ABORT TERMA".

Whenever there is an error detected, an error message
identifying the error and error code is printed onto 
standard I/O and the processor is then halted by using 
the 'halt' primitive. Here is a list of the possible 
errors:

- file 'TCREG.TXT' could not be reset properly.
- file 'TCRES.TXT' could not be reset properly.
- file 'TDREG.TXT' could not be reset properly.

If any errors are detected by one of the routines called, 
then the routine itself will print an error message and 
halt the processor.

REMARKS:
The files 'TCREG.TXT', 'TCRES.TXT' and 'TDREG.TXT' are 
formatted in accordance with the specifications of this 
module. Any changes in the format of this module or 
of those text files should be done very carefully.

************************************************************************

************************************************************************
NAME: TINIT

FUNCTION: This module is used to create a dynamic region for task TERMA by issuing a CRRO$ memory management directive.

USAGE: procedure tinit(var id: integer); external;

  tinit(id);

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: id - the region id.

MODULES CALLED: (The symbol '>:' designates a primitive)

  >halt
  >writeln
  >rrg'

EXIT CONDITIONS:
If there is an error detected in calling CRRO$, then an error message identifying the error and error code is printed onto standard I/O and the processor is then halted by using the 'halt' primitive.

The return code of the CRRO$ routine is set to greater or equal to 0 (zero) in case of successful completion and to less than 0 in case of unsuccessful completion.

For more information on the use of CRRO$ and of its return codes, see pages 6-36, 6-38 of the RSX-11M V3.2 Executive Reference Manual.

REMARKS: At the moment, there is no protection on the region. One can easily remedy this by giving an appropriate value to the region protection word.
NAME:  TPRENTA (MAIN)

FUNCTION: To serve as Transport Protocol Entity for USER 'A'.

USAGE: from MCR: 'RUN TPRENTA /TASK TPRENTA /INC=100.'

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: NONE

MODULES CALLED: (The symbol '>' designates a primitive)

>chr
>new
>ord
>write
>writeln
>buffclear
>buffdo
>buffwait
>initial
>inspltdu
>prenptdu
>recca
>sendd
>werra
>wterm

EXIT CONDITIONS:
This program has been designed to run indefinitely until the task into which it runs is specifically aborted with the MCR command 'ABORT TPRENTA'.

If an error is detected by one of the routines called, then the routine itself will print an error message and halt the processor by using the 'halt' primitive.

REMARKS: The format of the TPDUs is different from the standard format since the PDP-11 swaps the Hi and LO bytes of a word in memory. So internally speaking (in memory) the
NAME: TSEND

FUNCTION: To send a buffer of information to task TPRENTA, by using the send by reference 'directive SREF$.

USAGE: procedure tsend(swndno: stdf
var wndno: wndno;
   id: integer) external;

   send(swndno, wndno, id)

INPUT PARAMETERS: swndno - the buffer to send by reference.
   wndno - the window number to be used (1 to 7).
   id - the region id.

OUTPUT PARAMETERS: wndno - the new window number.

MODULES CALLED: (The symbol '=>' designates a primitive)
   >halt
   >write
   craf
   sref

EXIT CONDITIONS: If there are any errors detected in calling CRAW$ or SREF$, then an error message identifying the error and error code is printed onto standard I/O and the processor is then halted by using the 'halt' primitive.

The return code of both these system directives is set to greater or equal to 0 (zero) in case of successful completion and to less than 0 in case of unsuccessful completion.

For more information on the use of CRAW$ and SREF$ and their return codes, see pages 6-31, 6-34 and 6-138, 6-140 of the RSX-11M V3.2 Executive Reference Manual.
REMARKS:
The method used in this module involves putting information in the region windows and sending these by reference.
Each call to this routine involves sending the next window in line (i.e., the next window in the region or window number one if the last window (i.e., nu 7) was used on the previous call). This strategy guarantees that at least 7 'SENDUs' can be done before any of the data sets overwritten by new data.
NAME:   VAL

FUNCTION: This function takes two characters c1(LO) and c2(HI) and
          returns the integer that the combination of HI-LO forms.

USAGE:   i : integer;

          function val(c1 : char; c2 : char) : integer; extern;

          i := val(c1, c2);

INPUT PARAMETERS:  c1 - the LO character,
                   c2 - the HI character.

OUTPUT PARAMETERS: (returned) - the integer formed by HI-LO.

MODULES CALLED:    (The symbol '>' designates a primitive)
                   halt
                   writeln

EXIT CONDITIONS:   Normal exit conditions (top-down)

REMARKS:           One should be aware that the hi and low bytes of a word
                   are positioned as LO-HI in memory on the POP-11.
NAME: WERRA

FUNCTION: Write an appropriate error message on User A's terminal due to an error situation in TPRENTA.

USAGE: procedure werra(pstrnd : pointer;
                          rjose : integer;
                          mes : wmessXerr); external

werra(pstrnd, rjose, mes);

INPUT PARAMETERS: pstrnd - Pointer to received TPDU,
                   rjose - Cause for rejection of TPDU,
                   mes - The error message number.

OUTPUT PARAMETERS: NONE

MODULES CALLED: (The symbol '*' designates a primitive)
                 *ord
                 *writeln

EXIT CONDITIONS: Normal exit conditions.

REMARKS: The TPRENTA module should be studied for a better understanding of the error encountered.

**********************************************************************************************
**NAME:** WHESS

**FUNCTION:** To write a NTRNSDA error message on the terminal in case illegal requests are made by TPRENTA or NTRNSDB.

**USAGE:**
```pascal
procedure whess(messno : integer;
              stateXa : state;
              rrea : netXrea): extern;
```

**INPUT PARAMETERS:**
- `messno`: the type of message.
- `stateXa`: the state of NTRNSDA.
- `rrea`: the inputed string.

**OUTPUT PARAMETERS:** None

**MODULES CALLED:** (The symbol '->' designates a primitive)
- `halt`
- `writeln`

**EXIT CONDITIONS:** Normal exit conditions.

**REMARKS:** None

**************************************************************
(2$w-1)

NAME: WQAB

FUNCTION: To send information to task MTRNSDB via system directives
          send data (SDAT$) or send by reference (SREF$).

USAGE: procedure waab(wtr wnd, wnd, nXrea, id, bol); extern

waab(wtr wnd, wnd, nXrea, id, bol);

INPUT PARAMETERS: wtr  - the pointer to the buffer to be sent.
                  wnd  - the window number to be used if bol = false.
                  nXrea - the string to be sent.
                  id   - the region id.
                  bol  - the boolean switch.

OUTPUT PARAMETERS: wnd  - the next window number.

MODULES CALLED: (The symbol $ denotes a primitive)

halt
writeln
craw
sref
sdat

EXIT CONDITIONS:

If there are any errors detected in calling CRAW$, SDAT$
or SREF$, then an error message identifying the error and
error code is printed onto standard I/O and the processor
is then halted using the 'halt' primitive.

The return code for all these system directives is set to

power or equal to 0 (zero) in case of successful completion
and less than 0 in case of unsuccessful completion.
(******-

**NAME:** WTERMA

**FUNCTION:** Write an appropriate legal message on the terminal A which could be TC1ND, TD1ND, TNODI or TCCOH with appropriate parameter values for the above primitives, as per ISO Transport Protocol Specs. Apr. 1983.

**USAGE:**

```pascal
procedure wterm(a : std;
    mes : wordXs); extern
```

```pascal
wterm(a: std; mes);
```

**INPUT PARAMETERS:**
- `a`: the pointer to the TPDU received.
- `mes`: the message number.

**OUTPUT PARAMETERS:** NONE

**MODULES CALLED:** (The symbol "\*" designates a primitive)
- `>char`
- `>ord`
- `>writeln`
- `val`

**EXIT CONDITIONS:** Normal exit conditions.

**REMARKS:**

A lot of the code is for generating a nice display on a VT100 compatible terminal.

Most of the fields in a TPDU are outputed using the 'ord' primitive. This is because the fields are stored in byte (character) format.

The VAL function is used to associate two bytes together and form an integer.
APPENDIX D

PASCAL CODE OF ROUTINES
OF THE PROTOTYPE FOR
CLASS-0 TP SPECIFICATIONS
(User A's side only)
NAME: BUFFCLEAR

FUNCTION: To clear the event flag corresponding to the passed integer value.

USAGE: procedure buffclear(efn : efn);

buffclear(efn);

INPUT PARAMETERS: efn - the event flag number.

OUTPUT PARAMETERS: NONE

MODULES CALLED: halt

<write>

ceref

EXIT CONDITIONS:

If the retcode from the CLEF$ Directive is less than zero then an error message is printed and the event flag may not have been cleared and the processor is halted using the 'halt' primitive.

REMARKS: See RSX-11M V3.2 Executive Directives Manual on page 6-26 for more information.

******************************************************************************

type 

efn = integer;

procedure buffclear(efn : efn);

var ids : integer;

procedure ciref(var efn : integer; var ids : integer); extern(fortran);
begin
{ writeln(tt, 'entering buffclear routine') }

clref(efn, ids); (* call CLEF *)
if ids < 0 then (* check if retcode is less than 0 *)
begin
writeln('error in using clref. Error code =', ids);
halt
end

{ writeln(tt, 'leaving buffclear routine') }
end.

NO ERROR DETECTED
TOTAL PROGRAM SIZE 000256
OUTERMOST DATA SIZE 000002
RESERVED STACK & HEAP 000522
COMPILATION TIME 6 SECONDS
BUFFGO.PAS

**NAME:** BUFFGO

**FUNCTION:** To set the event flag corresponding to the passed integer value.

**USAGE:**

```pascal
procedure buffgo(efn: efnn);
```

**INPUT PARAMETERS:**

- `efn` - the event flag number.

**OUTPUT PARAMETERS:**

- `efnn` - the event flag number.

**MODULES CALLED:**

- `halt`
- `writeln`
- `setef`

**EXIT CONDITIONS:**

- Normal exit conditions.
- If the retcode from the `SETEF` directive is less than zero then an error message is printed and the event flag may not be set and the processor is halted using the `halt` primitive.

**REMARKS:**

- See RSX-11M V3.2 Executive Directives Manual on page 6-123 for more information.
- Also see chapter 3 of the Manual for more info. on event flags and their use.

```pascal
type
  efnn = integer;

procedure buffgo(efn : efnn);
```

```pascal
var
  ids : integer;
```
procedure setef(var efn: integer; var ids : integer); extern(fortran);
begin
{ writeln(tty, 'entering buffdo routine'));
setef(efn, ids); (* call the SETF$ Directive *)
if ids < 0 then (* check if retcode is less than 0 *)
begin
  writeln(' error in setef. Error code = ', ids);
  halt
end
{ writeln(tty, 'leaving buffdo routine')};
end.

NO ERROR DETECTED

TOTAL PROGRAM SIZE 000250
EXTERNAL DATA SIZE 000002
RESERVED STACK & HEAP 000522
COMPILATION TIME 5 SECONDS

PAS BUFFDO,BUFFDO/P42=BUFFDO
BUFFWAIT

NAME:       BUFFWAIT

FUNCTION:   To wait until the event flag corresponding to the passed integer value
            is set.

USAGE:      procedure buffwait(efn: wfn);

            buffwait(efn);

INPUT PARAMETERS:  efn  - the event flag number.

OUTPUT PARAMETERS:  NONE

MODULES CALLED:
                    halt
                    writeln
                    waitfr

EXIT CONDITIONS:  Normal exit conditions.

            if the recode from the WTSE$ Directive is less
            than zero then an error message is printed and:
            the processor is halted using the 'halt' primitive.

REMARKS:
            See RSX-11M V3.2 Executive Directives Manual on page 6-168 for more
            information.

            One should be aware of the different types of event flags that
            can be used for different purposes since for example if global
            event flags are used for controlling events in a particular task
            then one must make sure that these flags are not used somewhere
            else in the system (see Chapter 3 of Manual).

*****************************************************************************

type
    efn = integer;
procedure buffwait(efn : efn);  
var   ids : integer;
procedure waitfr(var efn:integer; var ids : integer); extern(fortran);
begin
  writeln(tty, 'entering buffwait routine');
  waitfr(efn, ids);  (* wait for flag to be set *)
  if ids < 0 then begin (* was it successful? *)
    writeln('error in waitfr. Error code = ', ids);
    hult
  end
  writeln(tty, 'leaving buffwait routine');
end.

NO ERROR DETECTED.
TOTAL PROGRAM SIZE 000252
OUTERMOST DATA SIZE 000002
RESERVED STACK & HEAP 000522
COMPILATION TIME 7 SECONDS

PAS BUFFWAIT, BUFFWAIT/P42-BUFFWAIT
(***a-*)

(***********************************************************************

NAME:      FILLB

FUNCTION:  This procedure converts an integer to character format into the
            given character field.

USAGE:     procedure fillb(b: integer;
            var c: char); external

fillb(b, c);

INPUT PARAMETERS:  b        - the integer to convert.

OUTPUT PARAMETERS: c        - the character field to be assigned.

MODULES CALLED:   (The symbol '->' designates a primitive)

>chr
>halt
>writeln

EXIT CONDITIONS:  If the passed integer is > 256 or < 0 then
                  an error message is printed on the terminal
                  and the processor is halted.

REMARKS:        For Modularity and Portability reasons the primitive
                 'chr' should have been replaced by fillb in all the
                 modules in this system. This should definitely be
                 considered for future enhancements.

(***********************************************************************

procedure fillb(b: integer;
            var c: char); begin
{ writeln(tty, 'entering fillb subroutine'); } if ((b > 256) or (b < 0)) then
begin
writeln(tty, 'illegal passed integer in fillb routine');
halt
end;
c := chr(b)
{ writeln(tty, 'leaving fillb subroutine'); }
end.

NO ERROR DETECTED
TOTAL PROGRAM SIZE 000264
OUTERMOST DATA SIZE 000002
RESERVED STACK & HEAP 000520
COMPILATION TIME 6 SECONDS

PAS FILLB;FILLB/P42-FILLB
NAME:       FILLW

FUNCTION:  This procedure converts the passed integer to its equivalent
            two characters LO and HI.

USAGE:     procedure fillw( b : integer;
            var c1 : char;
            var c2 : char); external

            fillw(b, c1, c2);

INPUT PARAMETERS:  b - the integer to convert.

OUTPUT PARAMETERS: c1 - the LO field of the integer.
                   c2 - the HI field of the integer.

MODULES CALLED:   (The symbol '>.' designates a primitive)
                   >halt
                   >writeln

EXIT CONDITIONS:  Normal exit conditions

REMARKS:       It is to note here that the PDP-11 stores an integer's
                HI and LOW bytes in inverse order in memory. That is
                LO=HI.

procedure fillw( b : integer;
            var c1 : char;
            var c2 : char);

type
  trick - record
    (8 variant part $)
case bool: boolean of
    false: ( w: integer );
    true: ( c: packed array [1..2] of char );
end;

var
    tt: trick;
begin
    writeln(tt, 'entering fillw subroutine');
    tt.bool := false;
    tt.w := bi (% put integer in integer part of variant record %)
    tt.bool := true;
    c1 := tt.c[1]; (% separate the integer into 2 character fields %)
    c2 := tt.c[2];
    writeln(tt, 'leaving fillw subroutine');
end.
NAME: INITIAL

FUNCTION: This module is used to create a dynamic region for a task using the CRRG$ memory management directive.

USAGE: procedure initial(var id: integer); external

initial(id);

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: id - the region id.

MODULES CALLED: (The symbol '>' designates a primitive)

>halt
>writeln
>crrd

EXIT CONDITIONS: If there is an error detected in calling CRRG$, the error message identifying the error and the code is printed onto standard I/O and the processor is then halted by using the 'halt' primitive.

The return code of the CRRG$ routine is set to greater or equal to 0 (zero) in case of successful completion and to less than 0 in case of unsuccessful completion.

For more information on the use of CRRG$ and the return codes, see pages 6-36, 6-38 of the RSX-11M V3.2 Executive Reference Manual.

REMARKS: At the moment, there is no protection on the region. One can easily remedy this by giving an appropriate value to the region protection word.
type
  std = record
    fixp : packed array [1..FIXPLEN] of char;
    uth : packed array [1..OUTHLEN] of char;
  end;

  rstd = "std"

  ctrl = array[0..7] of integer;  { format for CRG$ }

  netXreq = packed array[1..NETREQLEN] of char;

  ddd = record
    rew : netXreq;
    dum : array[1..DUMMYLEN] of integer;
  end;

  rdd = "ddd"

  tskk = record
    num : array [1..2] of integer;
  end;

  rbuff = record  { format for RCVD$ }
    num : tskk;
    req : netXreq;
    dum : array [1..DUMMYLEN] of integer;
  end;

  buff = record  { format for SDAT$ }
    rew : netXreq;
    dum : array [1..DUMMYLEN] of integer;
  end;

  rddd = record
    uk : array[1..2] of integer;
    req : netXreq;
    dum : array[1..DUMMYLEN] of integer;
  end;

  rdd = "rddd"
ruwbbl = record ( format for CRAW and RREF )
  87  ruarr : packed array[1..2] of char
  88  rvbadd : rstd
  89  rsiz : integer
  90  rresid : integer
  91  ruff : integer
  92  rlen : integer
  93  rstrats : integer
  94  rsrbuff : rddi
  95
  96  end
  97
  98  wdbb = record ( format for CRAW and SREF )
  99    warr : packed array[1..2] of char
 100   wbadd : rstd
 101    wiz : integer
 102    wresid : integer
 103    wuff : integer
 104    wlen : integer
 105    wstrats : integer
 106    wsrbuff : rddi
 107
 108  end
 109
 110  state = (CLOSED, OPEN, WFCC, WFTRESP, WFNC) ( states of TPRENTA )
 111
 112  oxxsts = (opens, closed, inprogress)
 113
 114  wxxxxsts = (zero, one, two, three, four)
 115
 116  wxxxxerr = (five, six, seven, eight, nine, ten, eleven,
 117           twelve, thirteen, fourteen, fifteen, sixteen, seventeen, eighteen,
 118           nineteen, twenty, twentyone)
 119
 120  idss = integer
 121
 122  efnn = integer
 123
 124  tagg = integer
 125
 126  tntt = integer
 127
 128  wndnno = 1..NOPTRI
 129
 130  arr = packed array[1..2] of char
procedure initial(var id : integer);

var
    control : control;
    ids : idss;

procedure crrs(var control : control; var ids : idss); extern (furdyn);

begin
    { writeln(tty, 'entering initial subroutine'); }

    (* create and attach to region with a region name of 'A2' *)
    (* control[0] region id returned *)
    control[1] := REGSZ;
    (* size of the region in 32-word blocks also returned *)
    control[2] := 2880;
    (* region name (2 words in Radix-50 format) *)
    control[3] := 0;
    (* or 0 for no-name. Name is 'A2' *)
    (* Name of the partition that contains the *)
    control[5] := 0;
    (* region. (2 word in Radix-50 format *)
    (* presently it is GEN *)
    (* or 0 for the partition in which the task is *)
    (* running *)
    control[6] := 35;
    (* Region status word (see page 3-14) also returned *)
    (* bits set: *)
        RS.ATT = 32
    RS.WRT = 2
        RS.RED = 1
    (* *)
    control[7] := 0;
    (* Region protection code *)
    (* everything is permitted see > 3-13 *)
    crrs(control, ids);
    { writeln(tty, 'id assigned to the created region = ', control[0]); }
    writeln(tty, 'size in 32-W blocks of the attached region = ', control[1]);
    writeln(tty, 'region status word = ', control[6]);
    writeln(tty, 'ids = ', ids);
    if ids < 0 then
        begin
            writeln(tty, 'error on crrs, error code = ', ids);
            halt
        end;
    id := control[0]; (* stick region id in appropriate return field *)
NO ERROR DETECTED
TOTAL PROGRAM SIZE     000450
OUTERMOST DATA SIZE    000400
RESERVED STACK & HEAP  000542
COMPILATION TIME 19 SECONDS

PAS INITIAL\MP42-INITIAL
MAIN.

NAME: INITBUF

FUNCTION: To initialize a TPDU with zeros in all of its fields.

USAGE: procedure initbuf(var swndno : std); external;

initbuf(swndno);

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: 
  - swndno - the record to be filled with zeros in all of its fields.
    (initialization).

MODULES CALLED:
  - ('\ indictment a primitive )
    \chr
    \writeln

EXIT CONDITIONS: Normal condition.

REMARKS:
  - The FILLB routine should probably have been used here instead of chr.

GLOBAL CONSTANTS

const
  FIXLEN = 8;
  OTHLEN = 130;

GLOBAL TYPE

type std = record
  fixp : packed array [1..FIXLEN] of char;
  uth : packed array [1..OTHLEN] of char;
end;

procedure initbuf(var swndno : std);
44 var i : integer;
45     j : integer;
46 begin
47     writeln(tty, 'entering initbuf routine');
48     for i := 1 to FIXPLEN do
49         swndno.fxp[i] := chr(0);
50     for j := 1 to OTHLEN do
51         swndno.oth[j] := chr(0);
52     writeln(tty, 'leaving initbuf routine');
53 end.
NAME: INSPTPDU

FUNCTION: To inspect a TPDU for erroneous parameter codes or parameter values and returns the length of the erroneous packet if an error is detected and sets valid to FALSE; otherwise sets valid to TRUE.

USAGE: procedure insptpdu(strwnd : pstdi);
    var valid : boolean;
    var rjcs : integer;
    var err : err);

insptpdu(strwnd, valid, rjcs, err);

INPUT PARAMETERS: strwnd - the pointer to the TPDU to inspect.

OUTPUT PARAMETERS: valid - set to FALSE if error is detected and to TRUE otherwise.
    rjcs - the detected cause for rejection of TPDU.
    err - the length of the erroneous string which is sent back.

MODULES CALLED: (The symbol '>' designates a primitive)
>ord
>writeln
val

EXIT CONDITIONS: If an error is detected in the received buffer, the length of the erroneous string is returned with valid = FALSE; otherwise valid is set to TRUE.

REMARKS: Length returned by this routine is up to and including the first erroneous parameter or STRLEN.
in case of successful inspection.

At present the inspection performed is not very sophisticated since it is basically done by comparing the different fields of a particular TPDU with some fixed global constants which represent the only value acceptable. For our purposes this method has proven to be sufficient.

******************************************************************************

{#1TPRENTA.DEF}
1 \textbf{const} \\
2 \texttt{FIXPLEN} = 8; \\
3 \texttt{OTHLEN} = 130; \\
4 \texttt{STRLN} = 118; \\
5 \texttt{NETREQLEN} = 6; \\
6 \texttt{CRCODE} = 224; \\
7 \texttt{CCCODE} = 208; \\
8 \texttt{UTCODE} = 240; \\
9 \texttt{DRCODE} = 128; \\
10 \texttt{NDRCODE} = 120; \\
11 \texttt{ERCODE} = 112; \\
12 \texttt{APRND} = 7; \\
13 \texttt{WNDSZ} = 3; \\
14 \texttt{REGSZ} = 21; \quad \texttt{HDPTR times WNDSZ} \\
15 \texttt{NOPTR} = 7; \\
16 \texttt{DUMLEN} = 10; \\
17 \texttt{DUMHLEN} = 5; \\
18 \texttt{MODATFLG} = 20; \\
19 \texttt{IVCODE} = 193; \\
20 \texttt{DVCODE} = 224; \\
21 \texttt{ERLEN} = 128; \\
22 \texttt{GDEST} = 222; \\
23 \texttt{GCSOUR} = 1111; \\
24 \texttt{TPDUSZPL} = 1; \\
25 \texttt{TPDUSZPC} = 192; \\
26 \texttt{THPC} = 137; \\
27 \texttt{GCPTDU} = 128; \\
28 \texttt{GCTHR1} = 1; \\
29 \texttt{GCTHR2} = 2; \\
30 \texttt{GCTHR3} = 3; \\
31 \texttt{GCTHR4} = 4; \\
32 \texttt{THPL} = 12; \\
33 \texttt{TRDLPL} = 8; \\
34 \texttt{TRDLPC} = 136; \\
35 \texttt{GCTR1} = 7; \\
36 \texttt{GCTR2} = 8; \\
37 \texttt{GCTR3} = 9; \\
38 \texttt{GCTR4} = 10; \\
39 \texttt{NREOT} = 99; \\
40 \texttt{GCRESO} = 11; \\
41 \texttt{UNDEF} = 99;
type
  std = record  (* GLOBAL TYPES *)
    fixp : packed array [1..FIXPLEN] of char;
    oth : packed array [1..OTHLEN] of char;
  end;
  rstd = "std";
  contrl = array[0..7] of integer; (* format for CRRG* *)
  netXrea = packed array[1..NETREQLEN] of char;
  ddd = record
    reu : netXrea;
    dum : array[1..DUHLEN] of integer;
  end;
  rdd = "ddd";
  tskk = record
    num : array [1..2] of integer;
  end;
  rbuff = record (* format for RCVD*)(
    num : tskk;
    req : netXrea;
    dum : array [1..DUHLEN] of integer;
  end;
  buff = record (* format for SDAT*)(
    reu : netXrea;
    dum : array [1..DUHLEN] of integer;
  end;
  rddd = record
    uk : array[1..2] of integer;
    req : netXrea;
    dum : array[1..DUHLEN] of integer;
  end;
  rdd = "rddd";

rwdbb = record ( format for CRAW$ and RREF$ )
  raddr : packed array[1..2] of char
  rvbadd : stdf
  rsiz : integer
  rresid : integer
  roffs : integer
  rlen : integer
  rstats : integer
  rsrbuff : stdf
end;

wdbb = record ( format for CRAW$ and SREF$ )
  arr : packed array[1..2] of char
  vbadd : stdf
  siz : integer
  resid : integer
  ooffs : integer
  len : integer
  wstats : integer
  srbuff : stdf
end;

state = (CLOSED, OPEN, WFC, WFTRESP, WFNC) ( states of TPRENTA )

opXsts = (opens, closed, inprogress);

waessXsts = (zero, one, two, three, four);

waessXerr = (five, six, seven, eight, nine, ten, eleven,
  twelve, thirteen, fourteen, fifteen, sixteen, seventeen, eighteen,
  nineteen, twenty, twentyone);

idss = integer;

efnn = integer;

tagg = integer;

init = integer;

wndnoo = 1..NOPTRI;

arr = packed array[1..2] of char
procedure insptpdo(ptrwnd : psddl;
    var valid : boolean;
    var rjcase : integer;
    var err : err);

var

i : integer;

function val(c1 : char; c2 : char) : integer; external

begin

{ writeln(tty, 'entering insptpdo subroutine') }
valid := true; (* valid until proven otherwise *)
err := ERRLEN; (* length of the error string *)
rjcase := 0; (* no cause specified *)

CASE ord(ptrwnd^fixp[1]) OF (* select the proper TPDU type *)
CRCODE:
begin
(* LI.field ??????????? *)
(* DESTINATION ADDRESS *)
if (val(ord(ptrwnd^fixp[3]), ptrwnd^fixp[4]) <> DCSOUR) then
begin
  err := 5;
  rjcase := 3;
  valid := false
end
else (* SOURCE ADDRESS *)
if (val(ord(ptrwnd^fixp[5]), ptrwnd^fixp[6]) <> GCDEST) then
begin
  err := 7;
  rjcase := 3;
  valid := false
end
else (* TPDUZPL *)
if (ord(ptrwnd^alth[7]) <> TPDUSZPL) then
begin
  err := 16;
  rjcase := 3;
valid := false;
end
else (* TPDUSZFC *)
if ord(plrwnd^*i^*o*th[8]) <> TPDUSZFC) then
begin
  err := 17;
  rjcsx := 1;
  valid := false
end
else (* THPC *)
if ord(plrwnd^*i^*o*th[9]) <> THPC) then
begin
  err := 18;
  rjcsx := 1;
  valid := false
end
else (* TPDUSIZE *)
if ord(plrwnd^*i^*o*th[10]) <> GCTPDU) then
begin
  err := 19;
  rjcsx := 3;
  valid := false
end
else (* THROUGHPUT 1 *)
if (val(plrwnd^*i^*o*th[11], plrwnd^*i^*o*th[12]) <> GCTHR1) then
begin
  err := 21;
  rjcsx := 3;
  valid := false
end
else (* THROUGHPUT 2 *)
if (val(plrwnd^*i^*o*th[13], plrwnd^*i^*o*th[14]) <> GCTHR2) then
begin
  err := 23;
  rjcsx := 3;
  valid := false
end
else (* THROUGHPUT 3 *)
if (val(plrwnd^*i^*o*th[15], plrwnd^*i^*o*th[16]) <> GCTHR3) then
begin
  err := 25;
  rjcsx := 3;
  valid := false
end
  else (* THROUGHPUT 4 *)
  if (val(ptrwnd,rth[17], ptrwnd,rth[18]) < GCTR4) then
    begin
      err := 27;
      rjcse := 3;
      valid := false;
    end
  end
  else (* THPL *)
  if (ord(ptrwnd,rth[23]) <> THPL) then
    begin
      err := 32;
      rjcse := 3;
      valid := false;
    end
  end
  else (* TRDLPL *)
  if (ord(ptrwnd,rth[24]) <> TRDLPL) then
    begin
      err := 33;
      rjcse := 3;
      valid := false;
    end
  end
  else (* TRDLPC *)
  if (ord(ptrwnd,rth[25]) <> TRDLPC) then
    begin
      err := 34;
      rjcse := 1;
      valid := false;
    end
  end
  else (* TRANSIT DELAY 1 *)
  if (val(ptrwnd,rth[27], ptrwnd,rth[28]) <> GCTR1) then
    begin
      err := 37;
      rjcse := 3;
      valid := false;
    end
  end
  else (* TRANSIT DELAY 2 *)
  if (val(ptrwnd,rth[29], ptrwnd,rth[30]) <> GCTR2) then
    begin
      err := 39;
      rjcse := 3;
      valid := false;
    end
  end
else (* TRANSIT DELAY 3 *)
if (val(ptrwnd^.oth[31], ptrwnd^.oth[32]) <> OCTR03) then
begin
    err := 41;
    ruse := 3;
    valid := false
end;
else (* TRANSIT DELAY 4 *)
if (val(ptrwnd^.oth[33], ptrwnd^.oth[34]) <> OCTR04) then
begin
    err := 43;
    ruse := 3;
    valid := false
end;
CCCODE:
begin
(* LI field ?????????? *)
(* SOURCE ADDRESS *)
if (val(ptrwnd^.fikp[5], ptrwnd^.fikp[6]) <> OCEDEST) then
begin
    err := 7;
    ruse := 3;
    valid := false
end;
else (* TPDUSZPL *)
if (ord(ptrwnd^.oth[7]) <> TPDUSZPL) then
begin
    err := 16;
    ruse := 3;
    valid := false
end;
else (* TPDUSZPC *)
if (ord(ptrwnd^.oth[8]) <> TPDUSZPC) then
begin
    err := 17;
    ruse := 1;
    valid := false
end;
else (* THFC *)
if (ord(ptrwnd^.oth[9]) <> THFC) then
begin
  err := 18;
  rcse := 1;
  valid := false
end
else (* TPDU SIZE *)
if (ord(ptrwnd$.oth[10]) <> GCTPDU) then
  begin
    err := 19;
    rcse := 3;
    valid := false
  end
else (* THROUGHPUT 1 *)
if (val(ptrwnd$.oth[11]", ptrwnd$.oth[12]) <> GCTHR1) then
  begin
    err := 21;
    rcse := 3;
    valid := false
  end
else (* THROUGHPUT 2 *)
if (val(ptrwnd$.oth[13]", ptrwnd$.oth[14]) <> GCTHR2) then
  begin
    err := 23;
    rcse := 3;
    valid := false
  end
else (* THROUGHPUT 3 *)
if (val(ptrwnd$.oth[15]", ptrwnd$.oth[16]) <> GCTHR3) then
  begin
    err := 25;
    rcse := 3;
    valid := false
  end
else (* THROUGHPUT 4 *)
if (val(ptrwnd$.oth[17]", ptrwnd$.oth[18]) <> GCTHR4) then
  begin
    err := 27;
    rcse := 3;
    valid := false
  end
else (* THPL *)
if (ord(ptrwnd$.oth[23]) <> THPL) then
  begin
272       err := 32;
273       rjcase := 3;
274       valid := false
275     end
276     else (* TRDPL *)
277     if (ord(ptrwnd.oth(24)) <> TRDPL) then
278       begin
279         err := 33;
280         rjcase := 3;
281         valid := false
282       end
283     else (* TRDPC *)
284     if (ord(ptrwnd.oth(25)) <> TRDPC) then
285       begin
286         err := 34;
287         rjcase := 3;
288         valid := false
289       end
290     else (* TRANSIT DELAY 1 *)
291     writeln((chr 'Lr1 ' val(ptrwnd.oth(27), ptrwnd.oth(28))));
292     if (val(ptrwnd.oth(27), ptrwnd.oth(28)) <> GCTR1) then
293       begin
294         err := 37;
295         rjcase := 3;
296         valid := false
297       end
298     else (* TRANSIT DELAY 2 *)
299     if (val(ptrwnd.oth(29), ptrwnd.oth(30)) <> GCTR2) then
300       begin
301         err := 39;
302         rjcase := 3;
303         valid := false
304       end
305     else (* TRANSIT DELAY 3 *)
306     if (val(ptrwnd.oth(31), ptrwnd.oth(32)) <> GCTR3) then
307       begin
308         err := 41;
309         rjcase := 3;
310         valid := false
311       end
312     else (* TRANSIT DELAY 4 *)
313     if (val(ptrwnd.oth(33), ptrwnd.oth(34)) <> GCTR4) then
314       begin
err := 43;
rjcsr := 3;
valid := false
end
end;

DTCODE:
begin
(* LI field ????????? *)

if (ord(ptrwnd^.fix[4]) <> NREOT) then
begin
  err := 5;
  rjcsr := 3;
  valid := false
end
else (* check if length of USER DATA is not too long *)

if (ord(ptrwnd^.util[1]) > STRLEN) then
begin
  err := 10;
  rjcsr := 3;
  valid := false
end
end;

DRCODE:
begin
(* LI field ????????? *)

(* DESTINATION ADDRESS *)

if (val(ptrwnd^.fix[3], ptrwnd^.fix[4]) <> GCSOUR) then
begin
  err := 5;
  rjcsr := 3;
  valid := false
end
else (* SOURCE ADDRESS *)

if (val(ptrwnd^.fix[3], ptrwnd^.fix[6]) <> GCDEST) then
begin
  err := 71;
  rjcsr := 3;
  valid := false
end
else (* REASON *)
if (ord(p_strb).fix(8)) <> GCRESO then
begin
  err := 91
  rcse := 31
  valid := false
end
else (* legel DVL *)
if (ord(p_strb).uth(1)) <> STRLEN then
begin
  err := 101
  rcse := 31
  valid := false
end
else (* DUCODE *)
if (ord(p_strb).uth(2)) <> DUCODE then
begin
  err := 111
  rcse := 31
  valid := false
end
end

OTHERS: (* all other codes *)
begin
  err := 21 (* error in code *)
  rcse := 21
  valid := false
end
end (* CASE *)

{ writeln(tty, 'leaving insptpd subroutine'); }
NAME: NTAINIT

FUNCTION: This module is used to create a dynamic region for
   task NTRNSDA by issuing a CRGO$ memory management directive.

USAGE: procedure ntainit(var id : integer); external
   ntainit(id);

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: id - the region id.

MODULES CALLED: (The symbol '>' designates a primitive)
   halt
   writeln
   crrd

EXIT CONDITIONS:
   If there is an error detected in calling CRGO$, then
   an error message identifying the error and the error
   code is printed onto standard I/O and the processor
   is then halted by using the 'halt' primitive.

   The return code of the CRGO$ routine is set to greater
   or equal to 0 (zero) in case of successful completion
   and to less than 0 in case of unsuccessful completion.

   For more information on the use of CRGO$ and of its
   return codes, see pages 6-36 to 6-38 of the RSX-11M V3.2

REMARKS: At the moment, there is no protection on the region.
One can easily remedy this by giving an appropriate
value to the region protection word.
const
  FIXPLEN  = 81
  OTHLEN  = 1301
  NETREQLEN  = 61
  APRND  = 71
  WNDSZ  = 31
  REGSZ  = 211  \(\text{NOPTR times WNDSZ}\)
  NOPTR  = 71
  DUMLEN  = 101
  DUMMLEN  = 31
  MODATFLG  = 211
  TERMAFLAG  = 421
  TERMFLG  = 401
  TRANSDAFLAG  = 431
  TRANSDFLAG  = 411
  CRCODE  = 2241
  CCCODE  = 2081
  ZRCCODE  = 1281
  DTCODE  = 2401
  ERCODE  = 1121

typ
  std = record
    fixp : packed array [1..FIXPLEN] of char
    uth : packed array [1..OTHLEN] of char
  end;

  *std = std;

  contrl = array[0..7] of integer;  \(\text{format for CRRE Executive Directive.}\)

  netXrea = packed array[1..NETREQLEN] of char

  ddd = record
    netXrea
    dum : array[1..DUMMLEN] of integer
  end;

  *ddd = ddd;

  tskk = record
    num : array [1..2] of integer
end;

rbuff = record
  lua : tskki
  rea : netreal
  dum : array [1..DUMLEN] of integer
end;

buff = record
  rew : netreal
  dum : array [1..DUMLEN] of integer
end;

rddd = record
  uk : array [1..2] of integer
  rea : netreal
  dum : array[1..DUMLEN] of integer
end;

rddd = "rddd"

rwbdb = record
  raw : packed array[1..2] of char
  rwbdd : rwbdb
  rwix : integer
  rresid : integer
  roffs : integer
  rlen : integer
  rwstats : integer
  rsrbuff : rddd
end;

wdbb = record
  arr : packed array[1..2] of char
  vbadd : rwbdb
  siz : integer
  rresid : integer
  offs : integer
  len : integer
  wstats : integer
  wsrbuff : rddd
end;

end.
state = (one, two, three, four);

idss = integer;
efnn = integer;
tadd = integer;
tnlt = integer;
wndnoo = 1..NOPTRI
procedure ntainit(var id : integer);
var
  control : contll;
  ids : idss;
procedure crrs(var control : contll var ids : idss) extern (fortran);
begin
  writeln(tty, 'entering ntainit subroutine');
  (* create and attach to region with region name 'A3' *)
  (* control[0] region id returned *)
  control[11] := REGSZ; (* size of the region in 32-word blocks also returned *)
  control[2] := 2920; (* region name (2 words in Radix-50 format) *)
  control[3] := 0; (* or 0 for no-name, name is 'A3' *)
  control[4] := 11414; (* name of the partition that contains the *)
  control[5] := 0; (* region, (2 word in Radix-50 format *)
  (* presently it is GEN *)
  (* or 0 for the partition in which the task is *)
  (* running *)
  control[6] := 35; (* region status word (see page 3-14) also returned *)
  (* bits set: *)
  RS.AIT = 32
  RS.WRT = 2
  RS.RED = 1
  control[7] := 0; (* region protection code *)
  crs(control, ids);
  writeln(tty, 'id assigned to the created region = ', control[0]);
  writeln(tty, 'size in 32-W blocks of the attached region = ', control[1]);
  writeln(tty, 'region status word = ', control[6]);
  writeln(tty, 'ids = ', ids);
  if ids < 0 then
    begin
      writeln(tty, 'error on crrs, error code = ', ids);
      halt;
    end;
  id := control[0]; (* stick region id in appropriate return field *)
NTAINIT

91  { writeln(tty, 'leaving ntainit subroutine'); }
92  end.

NO ERROR DETECTED
TOTAL PROGRAM SIZE   000656
OUTERMOST DATA SIZE  000002
RESERVED STACK & HEAP  000542
COMPILATION TIME 16 SECONDS

PAS NTAINIT, NTAINIT/P42-NTAINIT
NTAREC

NAME: NTAREC

FUNCTION: To wait until some information is sent to the NTNSDBA task via send data or send by reference and receives it via receive data or receive by reference.

USAGE: procedure ntarec(var ptrund : fstd;
                       var nxrea : netxrea;
                       var queue : boolean;
                       var messxlarenta : boolean); extern;

ntarec(ptrund, nxrea, queue, messxlarenta);

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS:

 ptrund - the pointer to receive by reference buffer,
 nxrea - receive data string or receive by reference string,
 queue - true if a receive data was performed
          false if a receive by reference was done,
 messxlarenta - true if data was sent by TPRENTA task
                false if data was sent by NTNSDBA task.

MODULES CALLED: (The symbol `'"' designates a primitive)

  chr,
  halt,
  writeln,
  buffwait,
  cwait,
  wnd,
  rwrite,
  receive,
  ref.

EXIT CONDITIONS: If there are any errors detected in calling CRAW, DTRG.
MARK$, RECEIVE$ or RREF$, then an error message identifying
the error and error code is printed onto standard I/O and
the processor is then halted by using the 'halt' primitive.

The return code of all these system directives is set to
greater or equal to 0 (zero) in case of successful completion
and to less than 0 in case of unsuccessful completion.

For more information on the use of these system routines
and their return codes, see the RSX-11M V3.2 Executive

REMARKS: When there is a receive by reference, the newly attached region
is detached in order to prevent depletion of the memory pool.

*$ITPRENTA.DEF$
const FIXPLEN = 81
OTHLEN = 1301
STRLLEN = 1181
METRELLEN = 61
CRCODE = 2241
CCCODE = 2081
DTCODE = 2401
OCRDEC = 1281
NRCODE = 1201
ERCODE = 1121
APRND = 71
WNDsz = 31
REGSZ = 217
NOPTR = 71
DUMLEN = 101
DUMMLLEN = 51
NODATFLG = 201
IVCODE = 1931
DUCODE = 2241
ERRLEN = 1281
GCDEGT = 22221
GCSOUR = 11111
TPDUSZPL = 11
TPDUSZPC = 1921
THPC = 1371
GETPDU = 1281
GCTHR1 = 11
GCTHR2 = 21
GCTHR3 = 31
GCTHR4 = 41
THPL = 121
TROKFL = 81
TRDPC = 1361
GCTR1 = 71
GCTR2 = 81
GCTR3 = 91
GCTR4 = 101
NREOT = 991
GCRESO = 1111
UNDEF = 991
type
  std = record
    fixp: packed array [1..FIXPLEN] of char;
    oth: packed array [1..OTHLEN] of char;
  end;
end;

{GLOBAL TYPES}

std = "std"

ctrl = array[0..7] of integer; (format for CRGB)

netXrea = packed array[1..NETREQLEN] of char;

ddd = record
  reu: netXrea;
  dum: array[1..DUMLEN] of integer;
end;

add = "ddd"

tskk = record
  nm: array [1..2] of integer;
end;

rbuff = record (format for RCVD)
  nm: tskk;
  rea: netXrea;
  dum: array [1..DUMLEN] of integer;
end;

buff = record (format for SND)
  reu: netXrea;
  dum: array [1..DUMLEN] of integer;
end;

rddd = record
  ok: array[1..2] of integer;
  rea: netXrea;
  dum: array[1..DUMLEN], of integer;
end;

pddd = "rddd"
rwdbb = record ( format for CRAW and RREF )
  arr : packed array[1..2] of char
  rvbadd : rstdi
  rsiz : integer
  rresid : integer
  roffs : integer
  rlen : integer
  rwstls : integer
  rsrbuf : rddi
end;

wdbb = record ( format for CRAW and SREF )
  arr : packed array[1..2] of char
  vbadd : rstdi
  siz : integer
  resid : integer
  offs : integer
  len : integer
  wstls : integer
  srbuf : rddi
end;

state = [CLOSED, OPEN, WFCC, WFTRESP, WFNFC]; ( states of TPRENTA )

opXsts = [opens, closed, inprogress];

waessXsts = [zero, one, two, three, four];

waessXerr = [five, six, seven, eight, nine, ten, eleven, twelve, thirteen, fourteen, fifteen, sixteen, seventeen, eighteen, nineteen, twenty, twentyone, twentytwo];

idss = integer

efms = integer

tags = integer

tntt = integer

wndnno = 1..NOPTRI

arr = packed array[1..2] of char
procedure buffwait(efn : efnni); extern;
procedure ntarec(var ptrwdb : pstdi
  var nXrea: netXrea;
  var queue : boolean;
  var messXlrenta : boolean);
var
  dummy : efnni;
efn : efnni
  sotit : boolean;
ids : idss;
irdb : contrli;
isrb : ruddi;
rbuf : rbuffi;
rwdb : rwbdbi;
rsrdb : rsrdbi;
tso : tssdi;
tnt : tntti;
ts : tskki;
procedure cswz(var rwdb : rwbdbi var ids : idss); extern(fortran);
procedure dtsr(var irdb : contrli var ids : idss); extern(fortran);
procedure mark(var efn : efnni
  var lad : laddi;
  var tnt : tntti;
  var ids : idss); extern(fortran);
procedure receiv(var tsk : tskki
  var rbuff : rbuffi
  var dummy : efnni
  var ids : idss); extern(fortran);
procedure rref(var rwdb ; rwbdbi
  var isrb : ruddi
  var ids : idss); extern(fortran);
begin
  writeln(tty, 'entering ntarec procedure');
messXtreinta := false; (* initialize booleans *)
queue := false;
slot := false;
efn := NODATFLG; (* initialize local event flag *)
while not slot do (* do until something is received *)
  begin

  (* receive the data *)
  receive(task, rbuf, dummy, ids); (* writeln(tty, 'receive ids = ', ids));
  if ids <= -8 then
    begin
      if ids < 0 then
        begin
          writeln(tty, 'error in receiving data using receive error code = ', ids);
          halt;
        end
      else
        begin
          nXreu := rbuf.real;
          slot := true;
          queue := true;
        end
    end
  end (* check if message was sent by treinta *)
  if ((buf.naa.naa[1] = 3258) and (buf.naa.naa[2] = 8561)) then
    messXtreinta := true
  end
  else (* receive by reference *)
    begin (* prepare to receive by reference *)

    (* window definition block *)
    .rwbdb.wstat := 0; (* window status word *)
    (* bits set: NONE *)
    ref(rwbdb, isrb, ids);
    writeln(tty, 'ref ids = ', ids);
    if ids <= -8 then
      begin
      if ids < 0 then
        begin
          writeln(tty, 'error in using ref, error code = ', ids);
          halt
        end
end

else

begin (* successful receive by reference *)

write(tt, 'ptr to buff = ', ord(rwdb, rrbuff))
write(tt, 'region id (pointer to attachment description) = ', ord(rwdb, rrregid))
write(tt, 'offset word specified by sender task = ', rwdb, roffs)
write(tt, 'length word specified by sender task = ', rwdb,rlen)
write(tt, 'window status word = ', rwdb,rwstats)
write(tt, 'virtual base address = ', ord(rwdb,rvbaddr))
write(tt, 'ids = ', ids))

sotit := true;

(* create receive window into the newly attached region *)

rwdb, rmap := chr(APRNO); (* Active page register *)

rwdb, rsiz := WNDSZ; (* the size of the window in 32-word blocks *)

rwdb, rrregid := rwdb, rrregid; (* region id *)

(* or 0 for task region *)

rwdb, roffs := rwdb, roffs; (* offset in region (32W blocks) at which

the window is to start mapping *)

rwdb, rlen := rwdb, rlen; (* specified only if WS.MAP is set *)

(* or 0 if the length is to default to either

the size of the window or the space remaining

in the region *)

rwdb, rwstats := 386; (* specified only if WS.MAP is set *)

(* window status word *)

(* bits set: *)

WS.64B = 256
WS.MAP = 128
WS.WRT = 2

write(rwdb, ids);

write(tt, 'window created ids = ', ids);

write(tt, 'id assigned to the window = ', ord(rwdb, rmap[1]));

write(tt, 'virtual base address = ', ord(rwdb, rvbaddr))

write(tt, 'length actually mapped in 32-word blocks = ', rwdb, rlen);

write(tt, 'window status word = ', rwdb, rwstats))

if ids < 0 then

begin

write(tt, 'error in using crwdr error code = ', ids))

halt

end;
(* put receive packet in returned buffer *)

(* receive *fix*) : rwdbrvbadd".fix*

(* put receive by reference control buffer into string field *)

(* detach from newly attached region *)

if rrdbrusions[0] = rrdbr.region id then

if rrdbr[6] = 0 if (* no bits set *)

if ids < 0 then

begin

write(tty, 'error trying to detach from newly attached region')

halt

end

(* check if message was sent by terrnta *)

if ((rrdbr.rsrbuff".uk[1] = 32858) and (rrdbr.rsrbuff".uk[2] = 8561)) then

begin

write(tty, 'message received from terrnta')

messxterrnta := true

end

end

else (* wait for a while and try again *)

begin

wait (* wait 10 *)

tnt := 2 (* seconds *)

waitfor end; tnt, ids)

buffwait(efn)

end

end

(* write (tty, 'leaving ntarec procedure') *)
NAME: NTRNSDA (MAIN)

FUNCTION: Network transducer for side 'A'.

USAGE: from MCR: 'RUN NTRNSDA /TASK=TRNSDA /INC-100.'

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: NONE

MODULES CALLED: (The symbol * designates a primitive)

buffclear
buffsave
buffsave
buffput
write
written
new
ord
chr

EXIT CONDITIONS:
This program has been designed to run indefinitely until the task into which it runs is specifically aborted with the MCR command "ABORT TRNSDA".

If any errors are detected by one of the routines called, then the routine itself will print an error message and halt the processor by using the 'halt' primitive.

REMARKS:
See Ref. (22) for more details.

Program ntrnsda(lty):
const
  FIXPLEN = 81;
  OTHLEN = 130;
  NETREOLEN = 61;
  APRNO = 71;
  WNDSZ = 31;
  REGSZ = 211;  (* NOPTR times WNDSZ *)
  NOPTR = 71;
  DUHLEN = 101;
  DUMMLEN = 51;
  NOBACLE = 211;
  TERMIFLAG = 421;
  TERMIFLAG = 401;
  TRNSDAFLAG = 431;
  TRNSDBFLAG = 411;
  CRCODE = 2241;
  CCCODE = 2061;
  DRCODE = 1281;
  DTCODE = 2401;
  ERCODE = 1121;

type
  std = record
    fixp: packed array [1..FIXPLEN] of char;
    oth: packed array [1..OTHLEN] of char;
  end;

  pstd = "std";

cntcl = array[0..7] of integer;  (* format for CRG6 Executive Directive *)

  netXreq = packed array[1..NETREOLEN] of char;

  ddd = record
    [reux: netXreq;
     dum: array[1..DUMMLEN] of integer;
    end;

  pdd = "ddd";

  tskk = record
    nam: array [1..2] of integer;
var
efn : efnn;
id : integer;
ids : idsi;
messXlprenta : boolean;
nXrea : netXrea;
strwnd : stdi;
queue : boolean;
rdbuf : netXrea;
rbuf : netXrea;
sdbuf : buffi;
srbuf : dddi;
stateXa : state;
tas : taddi;
tnt : tntti;
wid : char;
wndno : wndnosi;

procedure buffclear(efn : efnn); extern;
procedure buffdo(efn : efnn); extern;
procedure buffwait(efn : efnn); extern;
procedure ntainit(var id : integer); extern;
procedure ntarec(var strwnd : stdi; 
  var nXrea : netXrea; 
  var queue : boolean; 
  var messXlprenta : boolean); extern;
procedure sdtlprenta(strwnd : stdi; 
  var wndno : wndnosi; 
  nXrea : netXrea; 
  id : integer; 
  bol : boolean); extern;
procedure waab(strwnd : stdi; 
  var wndno : wndnosi; 
  nXrea : netXrea; 
  id : integer; 
  bol : boolean); extern;
procedure wmsg(messno : integer);
  stateXe : state;
  ddum : netXrea;  external
begin
  (* for display purposes on VT100 compatible terminals *)
  writeln(tty, chr(27), 'Z2',
  chr(27), '7070f', chr(27), '"0m",
  chr(27), '7120r', chr(27), '"7a', chr(27), '"A"');
  writeln(tty, /*'************'*/); writeln(tty, 'z2');
  writeln(tty, 'TASK NTRNSDA activated'); writeln(tty, 'z2');
  writeln(tty, /*'************'*/);
  writeln(tty, chr(27), '"0m"');
  wndno := 1; (* initialize sending window *)
  stateXe := one; (* initialize state of NTRNSDA *)
  new(prwnd); (* allocate space for TPDU *)
  ndinit(id); (* create dynamic region with name "A3" *)

  (* make sure the screen is clear *)
  writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty);

  while(1 = 1) do (* DO forever *)
  begin
    (* print out the state of the transducer *)
    write(tty, chr(27), '7120f', '"STATE < "');
    if (ord(stateXe) = 0) then
      writeln(tty, 'IDLE ')
    else if (ord(stateXe) = 1) then
      writeln(tty, 'OUT.CON.PEND ')
    else if (ord(stateXe) = 2) then
      writeln(tty, 'IN.CON.PEND ')
    else
      writeln(tty, 'DATA_TRANSFER');

    (* receive packet from receive by reference queue or by task queue *)
    ntrrec(prwnd, nXrea, queue, messXrea);
134 (* print out the type of primitive and/or TPDU received *)
135 if queue then
136   begin
137     writeln(tty, chr(27), [B]20f', 'RECEIVED : ', nXreq);
138     writeln(tty, chr(27), [9]120f', 'TPDU : NONE ')
139   end
140 else if not messXprenta then
141   begin
142     writeln(tty, chr(27), [B]820f', 'RECEIVED : ', nXreq);
143     write(tty, chr(27), [9]120f', 'TPDU : ');
144     if ord(strwmd'.fixp[1]) = CRCODE then
145       writeln(tty, 'CR'
146     else if ord(strwmd'.fixp[1]) = CCCODE then
147       writeln(tty, 'CC'
148     else if ord(strwmd'.fixp[1]) = DTCODE then
149       writeln(tty, 'DT'
150     else if ord(strwmd'.fixp[1]) = DRCODE then
151       writeln(tty, 'DR'
152     else if ord(strwmd'.fixp[1]) = ERCODE then
153       writeln(tty, 'ER'
154     else
155       writeln(tty, 'XX'
156   end
157 else
158   begin
159     writeln(tty, chr(27), [B]820f', 'RECEIVED :
160     write(tty, chr(27), [9]120f', 'TPDU : ');
161     if ord(strwmd'.fixp[1]) = CRCODE then
162       writeln(tty, 'CR'
163     else if ord(strwmd'.fixp[1]) = CCCODE then
164       writeln(tty, 'CC'
165     else if ord(strwmd'.fixp[1]) = DTCODE then
166       writeln(tty, 'DT'
167     else if ord(strwmd'.fixp[1]) = DRCODE then
168       writeln(tty, 'DR'
169     else if ord(strwmd'.fixp[1]) = ERCODE then
170       writeln(tty, 'ER'
171     else
172       writeln(tty, 'XX'
173   end;
174   writeln(tty, chr(27), [23]1f');
176 if (messXprenTa) then  (* the incoming message is from TPRENTA *)
177 begin  (* YES, *)
179 case stateXa of
180
181 (* if in idle state then do the following *)
182
184 one : begin
185 if (queue) then  (* receive data *)
187 begin
188 if (nXrea = 'ncrea ') then
189 begin
190 stateXa := two;  (* outXconnXpending *)
191 waub(#trwnd, wdnos, 'a', id, true)  (* SEND data to NTRNSDB *)
192 end
193 else  (* write an error message for debug purposes *)
194 waess(6, stateXa, nXrea)
195 end
196 else  (* if NOT queue then write for debug purposes *)
197 begin
199 writeln(tty, 'RECEIVE by ref. does not work in IDLE state');
200 writeln(tty, 'only strings such as NCREQ etc are accepted')
201 end
202 end  (* idle *)
203 (* if in outXconnXpending state then do the following *)
205
207 two : begin
208 if (queue) then  (* RECEIVE data *)
209 begin
210 if (nXrea = 'ndrea ') then
211 begin
212 stateXa := one;  (* back to idle state *)
213 waub(#trwnd, wdnos, 'd', id, true)  (* SEND data to NTRNSDB *)
214 end
215 else  (* for debug purposes *)
216 waess(6, stateXa, nXrea)
217 end
218 else  (* for debug purposes *)
begin
    writeln(tty, 'RECEIVE by ref. does not work in OUTCONNECT PENDING');
    writeln(tty, 'state; only strings such as NDREQ etc are accepted')
end
end

(*) outXconnXpending (*)

(* if in inXconnXpending state, then do the following *)

three : begin
if (queue) then
    (* RECEIVE data *)
begin
    if (nxrea = 'ndrea ') then
        begin
            stateXa := onei
            (* back to idle *)
            waab(strwmd, wndnr, 'd', id, true) (* SEND data to NTRNSDA *)
        end
    else if (nxrea = 'ncres ') then
        begin
            stateXa := fouri
            (* dataXtransferXrds *)
            waab(strwmd, wndnr, 's', id, true) (* SEND data to NTRNSDB *)
        end
    else
        (* erroneous net primitive sent by TPRENTA *)
        (* this situation will not arise *)
        waess(s, stateXa, nxrea)
    end
else
    (* if NOT queue then write for debug purposes *)
begin
    writeln(tty, 'RECEIVE by ref. does not work in INCONNECT PENDING');
    writeln(tty, 'state; only strings such as NCRES etc are accepted')
end
end

(*) INXCONNXPENDING (*)

(* if in dataXtransferXrds state, then do the following *)

four : begin
if (queue) then
    (* RECEIVE data *)
begin
    if (nxrea = 'ndrea ') then
        begin
            (* receive arguments *)
            waab(strwmd, wndnr, 'd', id, true) (* SEND data to NTRNSDA *)
        end
    else
        (* erroneous net primitive sent by TPRENTA *)
        (* this situation will not arise *)
        waess(s, stateXa, nxrea)
    end
else
    (* if NOT queue then write for debug purposes *)
begin
    writeln(tty, 'RECEIVE by ref. does not work in INCONNECT PENDING');
    writeln(tty, 'state; only strings such as NDREQ etc are accepted')
end
end

(*) INXCONNXPENDING (*)


```pascal
262 waab(ptrwnd, wndno, 'd', id, true)); (* SEND data to NTRNSDB *)
263 stateXa := one (* back to idle *)
264 else (* erroneous net primitive received *)
265 (* from TPREN *)
266 (* this situation will not arise *)
267 waess(& stateXa, nXrea)
268 end
269 else (* RECEIVE by ref. *)
270 begin (* concatenate a single TPDU in USER DATA field of a NET SERV. PRIHITIVES *)
271 waab(ptrwnd, wndno, 'nnod', id, false); (* SEND by ref. *)
272 end
273 (* RECEIVE by ref. *)
274 (* dataXtransferXfild *)
275 (* case *)
276 (* incoming user message *)
277 end
278 end
279 (*
280
281 the following section deals with the messages coming in from the
282 queueXbXtoXa
283 *)
284 else if (nXrea[1] = 'e') then
285 waess(? stateXa, nXrea)
286 else case stateXa of
287 (* if in idle state: then do the following *)
288 one:begin
289 if (queue) then (* RECEIVE data *)
290 begin
291 if (nXrea[1] = 'a') then
292 begin
293 sdtprenta(ptrwnd, wndno, 'ncmd', id, true); (* SEND data NCIND to TPREN *)
294 stateXa := three (* inXconnXpressing *)
295 end
296 else (* anything other than a 'a' *)
297 begin
298 waab(ptrwnd, wndno, 'e', id, true); (* SEND data to NTRNSDB *)
299 end
300 end
301 end
302 end
303 end
304 end
```


end

end

else (* if NOT queue then write for debug purposes *)

begin

writeln(tty, 'RECEIVE by ref, does not work in IDLE');

writeln(tty, 'state: only strings such as NCRES etc. are accepted')

end (* IDLE *)

(* if in outXconnXPending state, then do the following *)

two: begin

if (queue) then

begin

if (nXrea[1] = 'd') then

begin

wmsg(3, stateXa, nXrea);

stateXa := one (* back to idle *)

end

else if (nXrea[1] = 's') then

begin

sendXa(ptrwnd, wndno, 'NCCON ', id, true); (* SEND data NCCON to TPENTA *)

stateXa := four (* data transferXrdy *)

end

else

begin

waab(ptrwnd, wndno, 'e ', id, true) (* SEND data to NTRNSDB *)

end

end (* queue *)

else (* if NOT queue then write for debug purposes *)

begin

writeln(tty, 'RECEIVE by ref, does not work in OUTCONNECT PENDING')

writeln(tty, 'state: only letters such as 's' or 'd' are accepted')

end (* OUTXCONNXPENDING *)

(* if in inXconnXPending state, then do the following *)

three:

begin
if (queue) then  (* RECEIVE data from NTRNSDB *)
  begin
    if (nXrea[1] = 'd') then
    begin
      sdtprerta(ptrwnd, wndno, 'ndind', id, true);  (* SEND data NDIND to TPRENTA *)
      stateXa := one
    end
    else
    begin
      waab(ptrwnd, wndno, e', id, true)  (* SEND data to NTRNSDB *)
    end
  end
else  (* if NOT queue then write for debug purposes *)
  begin
    writeln(tty, 'RECEIVE by ref, does not work in INCONNECT PENDING')
    writeln(tty, 'state: only letters such as a 'd' are accepted')
  end
end;
(* inXconoXpending *)

(* if in dataXtransferXrdy state, then do the following *)

four: begin
  if (queue) then
  begin
    if (nXrea[1] = 'd') then
    begin
      sdtprerta(ptrwnd, wndno, 'ndind', id, true);  (* SEND data NDIND to TPRENTA *)
      stateXa := one  (* back to idle *)
    end
    else if (nXrea[1] = 'e') then
    begin
      waab(ptrwnd, wndno, e', id, true)  (* SEND data to NTRNSDB *)
    end
  end
else  (* RECEIVE by ref, *)
  begin
    if (nXrea = 'nnodr') then
    sdtprerta(ptrwnd, wndno, 'nnodr', id, false)
  (* SEND by ref, to TPRENTA, Disintegration of the net primitive i.e. NNODR with a TPDU in its user data field *)
391 separation
392 is not explicitly done but
393 the relevant portion of the
394 received net primitive
395 i.e. the TPDU is read by the
396 tpenta
397
398 else (* erroneous net primitive sent
399 by NTRNSDB. This situation will
400 not arise *)
401 begin
402 sendtpentaptrwnd, wndnr, 'ndnd', id, true); (* SEND data NDIND to TPRENATA *)
403 stateXa := one
404 end
405 end (* RECEIVE by ref. *)
406 end (* dataXtransferXrdy *)
407 end (* case *)
408 end (* end of RUN forever *)
409
410 writeln(tty, 'NTRNSDA TASK TERMINATED......................')
411 end.

NO ERROR DETECTED
TOTAL PROGRAM SIZE 013702
OUTERMOST DATA SIZE 000124
RESERVED STACK & HEAP 001152
COMPILATION TIME 100 SECONDS

PAS • NTRNSDA • NTRNSDA/P42•NTRNSDA
1  const  \{ GLOBAL CONSTANTS \}
2  \begin{verbatim}
3   FIXLEN  =  8;
4   OTHLEN = 130;
5   STRLEN = 110;
6   NETREGLEN = 61;
7      CRCODE = 2241;
8      CCCODE = 2081;
9      DTCODE = 2401;
10      DRCODE = 1281;
11      NDRCODE = 1201;
12      ERCODE = 1121;
13      APRNO = 71;
14      WNDsz = 31;
15      REGSZ = 211; \{ NOPTR times WNDsz \}
16      NOPTR = 71;
17      DUMLEN = 101;
18      DUMMLLEN = 51;
19      NODATFLG = 201;
20      IVCODE = 1931;
21      DVCODE = 2241;
22      ERRLEN = 1281;
23      GCDST = 22221;
24      GCSDUR = 11111;
25      TPDUSZPL = 11;
26      TPDUSZPC = 1921;
27      THPC = 13777;
28      GCTPDU = 1781;
29      GCTHR1 = 11;
30      GCTHR2 = 21;
31      GCTHR3 = 31;
32      GCTHR4 = 41;
33      THPL = 121;
34      TRDLP = 81;
35      TRDLPc = 1361;
36      GCTR01 = 71;
37      GCTR02 = 81;
38      GCTR03 = 91;
39      UL1RD3 = 101;
40      UL1RDU = 1111;
41      ULDSTU = 1111;
42      UN Hilton = 9991;
43  \end{verbatim}

type

{ GLOBAL TYPES }

std = record
  { TPDUs }
  fixp : packed array [1..FIXPLEN] of char;
  oth  : packed array [1..OTHLEN] of char;
end;

rstd = `std;

contr1 = array[0..7] of integer; { format for CRG$ }

netXreq = packed array[1..NETREQLEN] of char;

ddd = record
  reu : netXreq;
  dum : array[1..DUMHLEN] of integer;
end;

rddd = `ddd;

tskk = record
  nau : array [1..2] of integer
end;

rbuff = record { format for RCV$ }
  nau : tskk;
  reu : netXreq;
  dum : array [1..DUMHLEN] of integer;
end;

buff = record { format for SDAT$ }
  reu : netXreq;
  dum : array [1..DUMHLEN] of integer;
end;

rddd = record
  oia : array[1..2] of integer;
  reu : netXreq;
  dum : array [1..DUMHLEN] of integer;
end;

rddd = `rddd;

rddd = `rddd;
rwbdb = record ( format for CRAW$ and RREF$ )
  rarr : packed array[1..2] of char;
  rvbadd : rstd;
  rsiz : integer;
  rresid : integer;
  roffs : integer;
  rlen : integer;
  rwssts : integer;
  rsrbuf : rstd;
end;

wdbb = record ( format for CRAW$ and SREF$ )
  app : packed array[1..2] of char;
  vbadd : rstd;
  siz : integer;
  ssid : integer;
  ridid : integer;
  offs : integer;
  len : integer;
  wssts : integer;
  srbuf : rstd;
end;

state = (CLOSED, OPEN, WFC, WFTRESP, WFN); (states of TPRENTA)

opXsts = (opens, closed, inprogress);

wmsgXsts = (zero, one, two, three, four);

wmsgXerr = (five, six, seven, eight, nine, ten, eleven,
  twelve, thirteen, fourteen, fifteen, sixteen, seventeen, eighteen,
  nineteen, twenty, twenty-one);

idss = integer;
efnn = integer;
tass = integer;
tntt = integer;
wndgo = 1..NOPTR;
arr = packed array[1..2] of char;
130
131     errr = integer;
```pascal
procedure preptpd (ptrwnd : pstdi;
var dstref : arrt;
var srcref : arrt;
var psrc : integer;
err : error;
exit : integer)

var
i : integer;
j : integer;
temp : stdi;

procedure fillw(b : integer;

var c1 : char;
var c2 : char); external;

begin

(* PREPTPD *)

{ writeln(tty, 'entering preptpd routine') };

{ select the correct type of TPDU }:

case cod of

CRCODE :

begin

'dstref[1] := ptrwnd^.fixp[3];
srcref[1] := ptrwnd^.fixp[5];

end;

CCCODE :

begin

ptrwnd^.fixp[2] := chr(42);
ptrwnd^.fixp[3] := dstref[1];

end;

DTCODE :

begin

ptrwnd^.fixp[2] := chr(2);

end;

end.
```
ERCODE : (* prepare ER-TPDOU *)
    begin (* ERCODE *)
      pstrnd".fixp[1] := chr(ERCODE);
      pstrnd".fixp[2] := chr(130); (* LI *)
      pstrnd".fixp[3] := dstref[1]; (* fill up the fixed part of *)
      pstrnd".fixp[4] := dstref[2]; (* the ER TPDU *)
      pstrnd".fixp[6] := chr(rcjse); (* fill the rejection cause field *)
      pstrnd".oth[2] := chr(LVCODE);
      (* fill up the variable part of the ER TPDU up to where the error was *)
      detected *)
      (* assume NTHLEN - 2 = ERRLEN *)
      temp.fixp := pstrnd".fixp;
      temp.oth := pstrnd".oth;
      if err >= FIXPLEN then
        begin
          for i := 1 to FIXPLEN do
            pstrnd".oth[i + 2] := temp.fixp[i];
          for i := (FIXPLEN + 1) to err do
            pstrnd".oth[i + 2] := temp.oth[i];
          for i := (err + 1) to ERRLEN do
            pstrnd".oth[i + 2] := chr(UNDEF)
        end
      else
        begin
          for i := 1 to err do
            pstrnd".oth[i + 2] := temp.fixp[i];
          for i := (err + 1) to FIXPLEN do
            pstrnd".oth[i + 2] := chr(UNDEF);
          for i := (FIXPLEN + 1) to ERRLEN do
            pstrnd".oth[i + 2] := chr(UNDEF)
        end
    end

DRCODE : (* prepare DR-TPDOU *)
    begin (* DRCODE *)
    end

NDRCODE : (* prepare complete DR-TPDOU *)
    begin (* DRCODE *)
135   pstrnd^fixp[1] := chr(DRCODE);
136   pstrnd^fixp[2] := chr(127);   (* LI *)
137   fillw(GCDEST, pstrnd^fixp[3], pstrnd^fixp[4]);
138   fillw(GCSOUR, pstrnd^fixp[5], pstrnd^fixp[6]);
139   pstrnd^fixp[8] := chr(GCRRESO);
140   pstrnd^oth[1] := chr(1);      (* DVL *)
141   pstrnd^oth[2] := chr(DVCODE));
142   pstrnd^oth[3] := chr(7);
143   end
144   end                           (* case *)
145
146   { writeln(tty, 'leaving preptdu routine'); }
147   end.

NO ERROR DETECTED
TOTAL PROGRAM SIZE       004424
OUTERMOST DATA SIZE      000002
RESERVED STACK & HEAP   000740
COMPILATION TIME       51 SECONDS

PAS PREPTDU,PREPTDU/P42-PREPTDU
NAME: REAUDPDA

FUNCTION: To read in a string from standard input into the appropriate field of a particular packet, record its length and put it into the correct length field of this packet.

USAGE: procedure readUdata(var w: std); external

readUdata(w);

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: w - the packet to be filled with a string and its associated length.

MODULES CALLED:
>break
>coin
>read
>readn
>retrnl
>fillb

EXIT CONDITIONS: Normal condition.

REMARKS:
The input line is truncated to STRLEN, so care should be taken not to exceed this value.

The user (TERMINAL A) should make sure that his/her terminal buffer is at least >= STRLEN. This can be checked by issuing the MCR command 'SET /BUF-TI:'. If the buffer is too small, then this can be altered by the MCR command 'SET /BUF-TI:200', if a buffer of 200 bytes is desired for example.

Unpredictable behavior results when non-printable characters such as escape or end-of-file characters are inputted by this module.
const  
  STRLEN = 118;
  FIXPLEN = 8;
  OTHLEN = 130;

type  
  std = record
    fixp : packed array [1..FIXPLEN] of char;
    othp : packed array [1..OTHLEN] of char;
  end;

procedure fillb(tb : integer);
  var cl : char; extern;
procedure readUdata(var w : std);

const  
  revide = '7m';
  video  = '0m';
  up     = '2a';

var  
  i : integer;
  len : integer;

begin  
  (* set input into reverse video *)
  writeln(chr(27), revide, chr(27), up);
  (* read the input line and record its length *)
  break(tty);
  readln(tty);
  i := 0;
  if not(eoln(ttyin)) then
    begin
      i := 1;
      (* length of string must not exceed STRLEN *)
      while (not(eoln(ttyin)) and (i <= STRLEN)) do
        begin
          read(tty, w.othp[2*i]);
          i := i + 1
        end;
  end;
LEN := 1 - 1;

else (% empty line %)
LEN := 1;

(% put the length in the appropriate fields %)
fillb(len, w.th[11]);

(% put back in normal display mode %)
writeln(CHR(27), video, CHR(27), up)
NAME: READERM

FUNCTION: To read a string from standard input and record its length.

USAGE: procedure request(var buffer: request; var len: integer); external

readerM(buffer, len);

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: buffer - the string buffer.
len - the length of the string buffer.

MODULES CALLED: >break
>readln
>read
>readln
>writeln

EXIT CONDITIONS: Normal condition.

REMARKS:
The input line is truncated to REQLEN, so care should be taken not to exceed this value.

It is not recommended to input non-printable characters using this module since unpredictable behavior can result when these strings are outputted.

const REQLEN = 61; (GLOBAL CONSTANT)

type request = array [1..REQLEN] of char;

procedure readerM(var buffer: request; var len: integer);
const
revideo = '7m';
video = '10m';
up = 'CA';

var i : integer;

begin
(* set input into reverse video *)
writeln(CHR(27), revideo, CHR(27), up);

(* read the input line and record its length *)
break(tty);
readln(tty);
i := 0;
if not(eoln(ttywin)) then

begin
  i := 1;
  (* length of string must not exceed RELEN *)
  while (not (eoln(ttywin)) and (i < RELEN)) do
    begin
      read(tty, buffer[i]);
      i := i + 1
    end;

  len := i - 1;
end
else (* empty line *)
  len := i;

(* put back in normal display mode *)
write(ln(CHR(27), video, CHR(27), up));

end.
NAME: RECCA

FUNCTION: To wait until some information is sent to the TPRENTA task via send data or send by references and receives it via receive data or receive by reference.

USAGE: procedure recca(var *trwrd : stdf;
 var nXreq : netXreq;
 var queue : boolean;
 var messXterm : boolean)) external

recca(*trwrd, nXreq, queue, messXterm);

INPUT PARAMETERS: None

OUTPUT PARAMETERS:
  *trwrd - pointer to received by reference buffer.
  nXreq - receive data or reference string.
  queue - true if a receive data was performed.
  messXterm - false if a receive by reference was done.
  messXterm - true if data was sent by TERMA task.
  messXterm - false if data was sent by NTRNSD task.

MODULES CALLED: (The symbol '>' designates a primitive)

>chr
>halt
>writeln
buffwait
craw
dtrn
mark
receiv
rref

EXIT CONDITIONS: If there are any errors detected in calling CRAW$, DTRG$,
HARK$, RECEIV$ or RREF$, then an error message identifying...
the error and error code is printed onto standard I/O and
the processor is then halted by using the 'halt' primitive.

The return code of all these system directives is set to
greater or equal to 0(zero) in case of successful completion
and to less than 0 in case of unsuccessful completion.

For more information on the use of these system routines
and their return codes, see the RSX-11M V3.2 Executive

REMARKS: When there is a receive by reference, the newly attached region
is detached in order to prevent depletion of the memory pool.

*******************************************************************************

(*I+TPRENTA.DEF)
1 \const
2 \ \begin{align*}
3 & \text{FIXPLEN} = 6; \\
4 & \text{GTHLEN} = 130; \\
5 & \text{STRLEN} = 118; \\
6 & \text{NETREQLEN} = 6; \\
7 & \text{CRCODE} = 224; \\
8 & \text{CCCODE} = 208; \\
9 & \text{DTCODE} = 240; \\
10 & \text{DRCODE} = 128; \\
11 & \text{NDRCODE} = 120; \\
12 & \text{ERCODE} = 112; \\
13 & \text{APRHO} = 7; \\
14 & \text{WNSZ} = 3; \\
15 & \text{REGSZ} = 21; \quad \begin{array}{l}
16 & \text{(NOPTR times WNSZ)} \\
17 & \text{NOPTR} = 7; \\
18 & \text{DUMLEN} = 10; \\
19 & \text{DUMMLEN} = 5; \\
20 & \text{NODATFLG} = 20; \\
21 & \text{IVCODE} = 193; \\
22 & \text{DPCODE} = 224; \\
23 & \text{ERRLEN} = 128; \\
24 & \text{GCDEST} = 2222; \\
25 & \text{GCSDUR} = 1111; \\
26 & \text{TPDUSZPL} = 1; \\
27 & \text{TPDUSZPC} = 192; \\
28 & \text{THPC} = 137; \\
29 & \text{GCTPDU} = 128; \\
30 & \text{GCTHR1} = 1; \\
31 & \text{GCTHR2} = 2; \\
32 & \text{GCTHR3} = 3; \\
33 & \text{GCTHR4} = 4; \\
34 & \text{THPL} = 12; \\
35 & \text{TRDLPL} = 8; \\
36 & \text{TRDLPC} = 136; \\
37 & \text{GCTRD1} = 7; \\
38 & \text{GCTRD2} = 8; \\
39 & \text{GCTRD3} = 9; \\
40 & \text{GCTRD4} = 10; \\
41 & \text{NCRED} = 991; \\
42 & \text{GCRESD} = 11; \\
43 & \text{UNDEF} = 99; \\
\end{array}
\end{align*}
type std = record (TPDUs)
  fixp: packed array [1..FIXPLEN] of char;
  oth: packed array [1..OTHLEN] of char;
end;

estd := std;

ctrl = array[0..7] of integer; < format for CRG$ >

netXrea = packed array[1..NETREGLEN] of char;

ddd = record
  reu: netXrea;
  dum: array[1..DUHLEN] of integer;
end;

rddd = ~ddd;

tskk = record
  num: array[1..2] of integer;
end;

rbuff = record < format for RCD$ >
  num: tskk;
  reu: netXrea;
  dum: array[1..DUHLEN] of integer;
end;

buff = record < format for SDAT$ >
  reu: netXrea;
  dum: array[1..DUHLEN] of integer;
end;

rdd = record
  ok: array[1..2] of integer;
  reu: netXrea;
  dum: array[1..DUHLEN] of integer;
end;

rddd = ~rddd;
```
87     rwdbb = record ( format for CRAW$ and RREF$ )
88         rarr : packed array[1..2] of char;
89         rbadd : stdi;
90         rsiz : inteder;
91         rregid : inteder;
92         roffs : inteder;
93         rlen : inteder;
94         rwstats : inteder;
95         rsvbuff : stdi;
96     end;
97
98     wdbb = record ( format for CRAW$ and SREF$ )
99         warr : packed array[1..2] of char;
100        wbadd : stdi;
101        wsize : inteder;
102        wregid : inteder;
103        woffs : inteder;
104        wlen : inteder;
105        wstats : inteder;
106        wsbuff : stdi;
107     end;
108
109     state = (CLOSED, OPEN, WFCC, WFTRESP, WFNC); { states of TPRENTA }
110
111     opXstst = (opends, closed, inprogress);
112
113     wmsgXstst = (zero, one, two, three, four);
114
115     wmsgXerr = (five, six, seven, eight, nine, ten, eleven,
116                  twelve, thirteen, fourteen, fifteen, sixteen, seventeen, eighteen,
117                  nineteen, twenty, twentyone);
118
119     idss = inteder;
120
121     efnn = inteder;
122
123     tmsg = inteder;
124
125     tntt = inteder;
126
127     wndnoo = 1..NOPTR;
128
129     arr = packed array[1..2] of char
```
procedure buffwait(efn : efnn); external

procedure recva(var etwmd : psld);
  var nXrea : netXrea;
  var queue : boolean;
  var messXterm : boolean);

var
dummy : efnn;
  efn : efnn;
  gotit : boolean;
  ids : idss;
  irdb : contrl;
  isrb : rddd;
  rbuf : rbuff;
  rwd : rwwdb;
  rrd : rwwdb;
  tms : lays;
  tnt : tntti;
  tsk : tskk;

procedure craw(var rwd : rwwdb; var ids : idss): external(fortran);

procedure dtrsv(var irdb : contrl; var ids : idss); external(fortran);

procedure mark(var efn : efnn;
  var tms : lays;
  var tnt : tntti;
  var ids : idss); external(fortran);

procedure receiv(var tsk : tskk;
  var rbuf : rbuff;
  var dummy : efnn;
  var ids : idss); external(fortran);

procedure prec(var rwd : rwwdb;
  var isrb : rddd;
  var ids : integer); external(fortran);

begin
  writeln(tty, 'entering recca procedure');

messXterm := false; (* initialize booleans *)
quen := false;
sqit := false;
efn := NOTAFILG; (* initialize waiting local event flag *)
while not sqit do (* do until something is received *)
begin

  (* receive the data *)
  receiv(tsk, rbuf, dummy, ids);
  writeln(ttw, 'receive ids = ', ids);
  if ids <> -8 then
  begin
    if ids < 0 then /*
      begin
        writeln(ttw, 'error in receiving data using receive error code = ', ids);
        halt
      end
    else /* receive by reference */
      begin /* prepare to receive by reference */
        (* window definition block *)
        rrwdb, rstatus := 0; (* window status word *)
        (* bits set: NONE *)
        reff(rrwdb, isrb, ids);
        writeln(ttw, 'reff ids = ', ids);
        if ids <> -8 then
          begin
            if ids < 0 then
              begin
                writeln(ttw, 'error in using reff error code = ', ids);
                halt
              end
            else /* successful receive by reference */
              begin
                (*
                  end
              end
            end
          end
        end
      end
    end
  end
```pascal
writeIn(tty, 'region id (pointer to attachment description = ', ord(wwdb, resid)));
writeIn(tty, 'offset word specified by sender task = ', wwdb, roffs);
writeIn(tty, 'length word specified by sender task = ', wwdb,rlen);
writeIn(tty, 'window status word = ', wwdb, rwstats);
writeIn(tty, 'virtual base address = ', ord(wwdb, rvbadd));

gotit := true;

(* create receive window into the newly attached region *)
wwdb, rprev[2] := chr(APRND); (* Active page register 7 *)
wwdb, rsiz := WNDSZ; (* the size of the window in 32-word blocks *)
wwdb, resid := wwdb, resid; (* region id *)
wwdb, roffs := wwdb, roffs; (* or 0 for task region *)
wwdb, roffs := wwdb, roffs; (* offset in region (32W blocks) at which
the window is to start mapping *)
wwdb, rlen := wwdb, rlen; (* specified only if WS.MAP is set *)
wwdb, rlen := wwdb, rlen; (* length to map (32W blocks) *)
wwdb, rwstats := 3861; (* or 0 if the length is to default to either
the size of the window or the space remaining
in the region *)
wwdb, rwstats := 3861; (* specified only if WS.MAP is set *)
wwdb, rwstats := 3861; (* window status word *)
wwdb, rwstats := 3861; (* size of the window or the space remaining
in the region *)
craw(wwdb, ids);
writeIn(tty, 'window created ids = ', ids);
writeIn(tty, 'id assigned to the window = ', ord(wwdb, rprev[1]));
writeIn(tty, 'virtual base address = ', ord(wwdb, rvbadd));
writeIn(tty, 'length actually mapped in 32-word blocks = ', wwdb, rlen);
writeIn(tty, 'window status word = ', wwdb, rwstats);
if ids < 0 then begin
  writeln(tty, 'error in using craw, error code = ', ids);
  halt;
end;

(* put receive packet in returned buffer *)
ptr wnd, fix := wwdb, rvbadd, fixp;
ptr wnd, 0th := wwdb, rvbadd, 0th;
```
192 (* detach from newly attached region *)
193 irdb[0] := rwbdb.resid; (* region id *)
194 irdb[6] := 0; (* no bits set *)
195 dtres(irdb, ids);
196 {
197 writeln(tty, 'ids of detach is ', ids);
198 if ids < 0 then
199 begin
200 writeln(tty, 'error trying to detach from newly attached region');
201 halt;
202 end;
203 (* check if packet is from terms *)
204 if ((rwbdb.rsrbuff.ek[1] = 32218) and (rwbdb.rsrbuff.ek[2] = 20840)) then
205 begin
206 writeln(tty, 'message from terms');
207 messxterm := true;
208 end;
209 end
210 end
211 (* wait for a while and try again *)
212 begin
213 wait := 10; (* wait 10 *)
214 tnt := 2; (* seconds *)
215 mark(efn, wait, tnt, ids);
216 buffwait(efn);
217 end
218 end
219 end
220 end
221 (* leaving reca procedure *)
222 end.

NO ERROR DETECTED
TOTAL PROGRAM SIZE 002472
OUTERHOST DATA SIZE 000002
RESERVED STACK & HEAP 000702
COMPILATION TIME 52 SECONDS

PAS 'RECA/RECA/P42=RECA
NAME: SDTPRENTA

FUNCTION: To send information to task TPRENTA via system directives
send data (SDAT$) or send by reference (SREF$).

USAGE: procedure sdtprenta(strwnd: pstdi;
   var wndno: wndno;
   nxrea: netxrea;
   id: integer;
   bol: boolean); extern

sdtprenta(strwnd, wndno, nxrea, id, bol);

INPUT PARAMETERS:
strwnd - pointer to the buffer to be sent,
wndno - the window number to be sent if bol = false,

nxrea - the string to be sent,
id - the region id,
bol - the boolean switch.

OUTPUT PARAMETERS:
wndno - the next window number.

MODULES CALLED: (The symbol 'r' designates a primitive)

rhalt
rwriteln
craw
sref
sdat

EXIT CONDITIONS:
If there are any errors detected in calling CRAW$, SDAT$
or SREF$, then an error message identifying the error and
error code is printed onto standard I/O and the processor
is then halted by using the 'halt' primitive.

The return code of all these system directives is set or
greater or equal to 0 (zero) in case of successful completion.
and less than 0 in case of unsuccessful completion.

For more information on the use of these system routines
and their return codes, see the RSX-11M V3.2 Executive

REMARKS:
See TSEND routine for information on method of circular
window queues and its restrictions.

************************************************************
PASCAL PDP-11  VERSION 6.07  1983-08-05  16:16:19  PAGE 3

TPRENTA.DEF

1 2  const  (GLOBAL CONSTANTS)

3 FIXPLEN = 81
4 DTHLEN = 130
5 STRLEN = 118
6 METRDLGEN = 61
7 CRCODE = 224
8 CCCODE = 208
9 DTCODE = 240
10 DRCODE = 128
11 NDRCODE = 120
12 ERCODE = 112
13 APRNO = 71
14 WNDsz = 31
15 REGsz = 211 (NOPTR times WNDsz)
16 NOPTR = 71
17 DUHLEN = 104
18 DUMMLEN = 51
19 MODATFLG = 20
20 IVCODE = 193
21 DVCODE = 224
22 ERRLEN = 128
23 GCDEST = 2222
24 GCSOUR = 1111
25 TPDUSZPL = 11
26 TPDUSZPC = 192
27 THPC = 137
28 GCTPDOU = 128
29 GCTHR1 = 11
30 GCTHR2 = 21
31 GCTHR3 = 31
32 GCTHR4 = 41
33 THPL = 12
34 TRDPL = 81
35 TRDLP = 136
36 GCTRDO = 71
37 GCTRDS = 81
38 GCTRDR = 91
39 GCTRDS = 101
40 NREDT = 99
41 OCRESO = 11
42 UNDEF = 99
PASCAL  FDP-11  VERSION  6.07  1983-08-05   16:16:19   PAGE  4
TPRENTA.DEF

  type
  ( GLOBAL TYPES )
  std = record
    ( TPDUs )
    fixp : packed array [1..FIXPLEN] of char;
    uth : packed array [1..OTHLEN] of char;
  end;

  pstd = "std;

  contr = array[0..7] of integer (.format for CRCG$ );

  netXreq = packed array[1..NETQDLEN] of char;

  ddd = record
    reu : netXreq;
    dum : array[1..DUHLEN] of integer;
  end;

  pdd = "ddd;

  tskk = record
    new : array [1..2] of integer;
  end;

  rbuff = record ( format for RCVD$ );
    tskk :
    reu : netXreq;
    dum : array[1..DUHLEN] of integer;
  end;

  buff = record ( format for SDAT$ );
    reu : netXreq;
    dum : array[1..DUHLEN] of integer;
  end;

  rddd = record
    ok : array[1..2] of integer;
    reu : netXreq;
    dum : array[1..DUHLEN] of integer;
  end;

  pddd = "rddd;

  rddd = "rddd;

  MAIN,
rwdbb = record  ( format for CRAW$ and RREF$ )
  rarr : packed array[1..23] of char
  rvbadd : rstdi;
  rsiz : integer;
  rresid : integer;
  roffs : integer;
  rlen : integer;
  rwstats : integer;
  rsrbuff : rstdi
end;

wdbb = record ( format for CRAW$ and SREF$ )
  arr : packed array[1..23] of char
  vbadd : rstdi;
  siz : integer;
  resid : integer;
  uoffs : integer;
  len : integer;
  wstats : integer;
  swbuff : rstdi
end;

state = (CLOSED, OPEN, WFCC, WFTRESP, WFNC); ( states of TPRENTA )
opXsts = (opens, closedx, inprogress);
waessXsts = (zero, one, two, three, four);
waaessXerr = (five, six, seven, eight, nine, ten, eleven, twelve, thirteen, fourteen, fifteen, sixteen, seventeen, eighteen, nineteen, twenty, twenty-one);
idss = integer;
efnn = integer;
tadd = integer;
tntt = integer;
wndnro = 1..NOPTRI;
arr = packed array[1..23] of char;
130  err = integer;
131  
procedure buffwait(efn: efnn); extern;

procedure buffclear(efn: efnn); extern;

procedure buffso(efn: efnn); extern;

procedure sdtprenta(ptrwd: pstdi;
   var mnu: mnuu;
   nxrea: netxrea;
   id: integer;
   bol: boolean);

var
   bf: buffi;
   efn: efnn;
   ids: idss;
   isrb: udd;
   swdb: wdbbi;
   tsk: tskki;
   wdb: wdbbi;

procedure crawl(var wdb: wdbbi; var ids: idss); extern(fortran);

procedure send(var tsk: tskki;
   var bf: buffi;
   var efn: efnn;
   var ids: idss); extern(fortran);

procedure sref(var tsk: tskki;
   var efn: efnn;
   var swdb: wdbbi;
   var isrb: udd;
   var ids: idss); extern(fortran);

begin

{ writeln(ttw, 'entering sdtprenta procedure'));

(* set up task name = 'TPRENA' *)

  tsk.name[1] := 32658;
  tsk.name[2] := 8561;

if bol then (* send data *)
begin
(* put string in buffer to be sent *)
bfr, len := nXref;

(* send the data to the 'TPRENA' task *)
send(tsk, bfr, len, ids);
{ writeln(tty, 'send ids = ', ids); }
if ids < 0 then
begin
  writeln(tty, 'error in calling SEND in SENDD, error code = ', ids);
  halt
end
else
begin (* send by reference to task 'TPRENA' *)

(* create a send window in the region *)
 wdb.apr[2] := chr(APRND); (* active page register 7 *)
 wdb.siz := WNDSZ;
 wdb.rsid := id;
 wdb.region := id;

(* or 0 for task region *)

(* specified only if WS.MAP is set *)

(* offset in region (32W blocks) at which
  the window is to start mapping *)

(* specified only if WS.MAP is set *)

(* length to map (32W blocks) *)

(* or 0 if the length is to default to either
  the size of the window or the space remaining
  in the region *)

(* specified only if WS.MAP is set *)

(* window status word *)

(* bits set: *)

WS.64B = 256
WS.MAP = 128
WS.WRT = 2

(* window created ids = ', ids); }

(* id assigned to the window = ', ord(wdb.apr[1]));

(* virtual base address = ', ord(wdb.vbaddr));

(* length actually mapped in 32-word blocks = ', wdb.len);

(* window status word = ', wdb.wstats); )
if ids < 0 then
begin

```pascal
144 writeln(tty, ' error in using cram: error code = ', ids);
145 halt
146 end;
147
148 (* stick buffer into the send window *)
149 wdb.vbadd$.fixp := ptrwnd$.fixp;
150 wdb.vbadd$.oth := ptrwnd$.oth;
151
152 (* prepare window definition block for send by reference *)
153 swdb.resid := id;
154 swdb.offs := (wndno-1) * WNDSZ;(* offset in region (32W blocks) *)
155 swdb.len := WNDSZ;(* length to map (32W blocks) *)
156 swdb:widths := 3;(* window status word *)
157 (* bits set:
158 WS.WRT = 2
159 WS.RED = 1
160 *)
161 sref(tsk, en, swdb, isrb, ids);
162 if ids < 0 then
163 begin
164 writeln(tty, ' error in using sref: error code = ', ids);
165 halt
166 end;
167
168 (* set up window number for next call *)
169 if wndno = NOPTR then
170     wndno := 1
171 else
172     wndno := wndno + 1;
173 end;
174
175 (* writeln(tty, ' leaving sdtprenta procedure')*)
176 end;

NO ERROR DETECTED
TOTAL PROGRAM SIZE 001704
OUTERMOST DATA SIZE 000002
RESERVED STACK & HEAP 000642
COMPILATION TIME 28 SECONDS
```
NAME: SEENDD

FUNCTION: To send information to task NTKNSDA via system directives.
           send data (SDAT$) or send by reference (SREF$).

USAGE: \texttt{procedure sendd(ptrwnd: \texttt{std};}
       \hspace{1em} var wndno: wndno;
       \hspace{1em} nXrea: netXrea;
       \hspace{1em} id: integer;
       \hspace{1em} bol: boolean;\texttt{;} extern;

       sendd(ptrwnd, wndno, nXrea, id, bol);

INPUT PARAMETERS: \texttt{ptrwnd} - the pointer to the buffer to be sent.
                  \texttt{wndno} - the window number to be used.
                  \texttt{nXrea} - the string to be sent if \texttt{bol} = true.
                  \texttt{id} - the region id.
                  \texttt{bol} - the boolean switch.

OUTPUT PARAMETERS: \texttt{wndno} - the next window number.

MODULES CALLED: (The symbol ‘$‘ designates a primitive)
</p>

\texttt{\textgreater\textgreater haltn \textgreater\textgreater writeln craf sref sdat}

EXIT CONDITIONS: If there are any errors detected in calling CRAW$, SDAT$ or SREF$, then an error message identifying the error and error code is printed onto standard I/O and the processor is then halted by using the 'halt' primitive.

The return code of all these system directives, is set or greater or equal to 0 (zero) in case of successful completion and less than 0 in case of unsuccessful completion.
For more information on the use of these system routines and their return codes, see the RSX-11M V3.2 Executive Reference Manual.

REMARKS: NONE

*******************************************************

{$ITPRENTA.DEF}
```
const
FIXPLEN = 8;
OTHLEN = 130;
STLEN = 110;
NETREQLEN = 6;
CRCODE = 224;
CCCODE = 208;
DTCODE = 240;
DRCODE = 128;
NDRCODE = 120;
ERCODE = 112;
APRND = 7;
WNDSZ = 31;
REGSZ = 21;
NOPTR = 7;
DHULEN = 10;
DUMMLEN = 5;
NODATFLG = 20;
IVCODE = 193;
DVCODE = 224;
ERRLEN = 128;
QCDEST = 2222;
GCSOUR = 1111;
TPDUSZPL = 11;
TPDUSZPC = 1923;
THFC = 137;
GCTPDU = 128;
GCTHR1 = 1;
GCTHR2 = 2;
GCTHR3 = 3;
GCTHR4 = 41;
THPL = 122;
TDLFL = 8;
TRDLFC = 136;
GCTRD1 = 7;
GCTRD2 = 8;
GCTRD3 = 9;
GCTRD4 = 10;
NREOT = 99;
GCRESO = 11;
UNDEF = 99;
```
type
  std = record
    fixp: packed array [1..FIXPLEN] of char;
    uhi: packed array [1..UHILEN] of char;
  end;

  pstd = "std;

  cntl = array[0..7] of integer; (* format for CRG$ *)

  netXrea = packed array[1..NETXRELEN] of char;

  ddd = record
    req : netXrea;
    dum : array[1..DUMLEN] of integer;
  end;

  rddd = ~ddd;

  tskk = record
    num : array [1..2] of integer;
  end;

  rbuff = record (* format for RCVD$ *)
    num : tskk;
    req : netXrea;
    dum : array [1..DUMLEN] of integer;
  end;

  buff = record (* format for SDAT$ *)
    req : netXrea;
    dum : array [1..DUMLEN] of integer;
  end;

  rddd = record
    ok : array[1..2] of integer;
    req : netXrea;
    dum : array[1..DUMLEN] of integer;
  end;

  rddd = ~rddd;
87 rwbdb = record ( format for CRAW and RREF )
88  rwbdb : packed array[1..2] of char
89  rsvadd : stdi
90  resiz : intederi
91  rresid : intederi
92  ruffs : intederi
93  rlen : intederi
94  rwbstats : intederi
95  rsrbuffer : stdi
96 end;
97
98 wdbb = record ( format for CRAW and SREF )
99  wdbb : packed array[1..2] of char
100  wvbadd : stdi
101  wsz : intederi
102  wresid : intederi
103  wuffs : intederi
104  wlen : intederi
105  wwbstats : intederi
106  wsrbuffer : stdi
107 end;
108
109 state = (CLOSED, OPEN, WFC, WFTRESP, WFNCH) ( states of TPRENTA )
110
111 opXsts = (opens, closed, inprogress);
112
113 waessXsts = (zero, one, two, three, four);
114
115 waessXerr = (five, six, seven, eight, nine, ten, eleven,
116              twelve, thirteen, fourteen, fifteen, sixteen, seventeen, eighteen,
117              nineteen, twenty, twenty-one);
118
119 idss = intederi
120
121 efnn = intederi
122
123 tags = intederi
124
125 tntt = intederi
126
127 wndnoo = 1..N0PTR;
128
129 arr = packed array[1..2] of char;
procedure send(dtrend : stdi);
  var wndnu : wndnuoi;
  nXrea : netXrea;
  id : integer;
  bol : boolean;

var
  bf : buffi;
  efni : efnni;
  ids : idssi;
  isrb : dddi;
  swdb : wdbbi;
  tsk : tskki;
  wdb : `wdbbi;

procedure crah(var wdb : `wdbbi var ids : idss); extern (fortran);

procedure send(var tsk : tskki;
  var bf : buffi;
  var efni : efnni;
  var ids : idss); extern(fortran);

procedure sref(var tsk : tskki;
  var efni : efnni;
  var swdb : wdbbi;
  var isrb : dddi;
  var ids : idss); extern(fortran);

begin
  writeln(`tentring sendd procedure');

(* set up task name = \`TRNSDA\')
  tsk,nam[1] := `32734;

if bol then (* send data *)
  begin
    (* put string in buffer to be sent *)
    bf,rev := nXrea;

    (* send the data to the `TRNSDA' task *)
    send(tsk, bf, efni, ids);
end.
```pascal
if ids < 0 then
  begin
    writeln(tty, 'error in calling SEND in SENDD, error code = ', ids);
    halt
  end
else
  begin
    (* create a send window in the region *)
    wdb.apr[2] := chr(APRND); (* Active page register 7 *)
    wdb.siz := WNDSZ;
    (* the size of the window in 32-word blocks *)
    wdb.resid := id;
    (* region id *)
    (* or 0 for task region *)
    wdb.offs := (windno-1) & WNDSZ; (* specified only if WS.MAP is set *)
    (* offset in region (32W blocks) at which
      the window is to start mapping *)
    wdb.len := WNDSZ;
    (* or 0 if WS.MAP is set *)
    (* length to map (32W blocks) *)
    wdb.wstats := 386;
    (* window status word *)
    (* or 0 if the length is to default to either
      the size of the window or the space remaining
      in the region *)
    (* specified only if WS.MAP is set *)
    if ids < 0 then
      begin
        writeln(tty, 'window created ids = ', ids);
        writeln(tty, 'Id assigned to the window = ', ord(wdb.apr[1]));
        writeln(tty, 'Virtual base address = ', ord(wdb.vbadd));
        writeln(tty, 'length actually mapped in 32-word blocks = ', wdb.len);
        writeln(tty, 'window status word = ', wdb.wstats);
      end
      halt
    end
    (* stick buffer into the send window *)
    wdb.vbadd^fix := ptrwnd^fix;
    wdb.vbadd^oth := ptrwnd^oth;
```
141 (* prepare window definition block to send by reference *)
142 swdb.resid := id;
143 (* id of the region to be sent by reference *)
144 swdb.offs := (wnrno-1)*WINDSZT; (* offset in region (32W blocks) *)
145 swdb.len := WINDSZT; (* length to map (32W blocks) *)
146 swdb.wstats := 3;
147 (* window status word *)
148 ; (* bits set: *)
149 ;
150 sref(tsk, efn, swdb, isrb, ids);
151 writeln(tty, 'ids for sref is - ', ids);
152 if ids < 0 then
153 begin
154 writeln(tty, ' error in using sref, error code = ', ids);
155 halt;
156 end;
157 (* set up window number to point to the next window in the circular queue *)
158 if wnrno = NOPT then
159 wnrno := 1
160 else
161 wnrno := wnrno + 1;
162 end
163 { writeln(tty, 'leaving sendd procedure');
164 end.
(*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*)

NAME: TERMA (MAIN)

FUNCTION: This module is used to continuously READ information from
USER A (TERMINAL A) and to send this information to the TPRENTER
 task. This information is obtained from terminal inputs in the
form of user primitive strings and also data strings (see struct
and thread primitives) and also from certain text files which
contain information related to the selected user primitive.

USAGE: from MCR: 'RUN TERMA /TASK=TERMA /INC=100.'

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: NONE

MODULES CALLED: (The symbol '>' designates a primitive)

>chr
>halt
>output
>read
>readin
>reset
>writeln
 buffsfree
 buffsread
 buffsread
 fillb
 fillw
 initbuf
 readterm
 readdata
 tinit
 tsend

EXIT CONDITIONS: This program has been designed to run indefinitely until
the task into which it runs is specifically aborted with
the MCR command 'ABORT TERMA'.

Whenever there is an error detected, an error message
identifying the error and error code is printed onto standard I/O and the processor is then halted by using the 'halt' primitive. Here is a list of the possible errors:

- file 'TCREQ.TXT' could not be reset properly.
- file 'TCRES.TXT' could not be reset properly.
- file 'TDREQ.TXT' could not be reset properly.

If any errors are detected by one of the routines called, then the routine itself will print an error message and halt the processor.

REMARKS:

The files 'TCREQ.TXT', 'TCRES.TXT' and 'TDREQ.TXT' are formatted in accordance with the specifications of this module. Any changes in the format of this module or of those text files should be done very carefully.

************

Program terma(tty, input);

(*ITERMA.DEF)
PASCAL PDP-11
TERMA.DEF

1  const
2
3  REQLEN  =  6;
4  N0PT0R  =  7;
5  MWDSZ   =  3;
6  APRNO   =  7;
7  REGSZ   =  2;
8  FIXPLEN =  8;
9  OTHLEN  =  130;
10  STRLEN  =  118;
11  CRCODE  =  22;
12  CCCCDE  =  28;
13  DTCDCE  =  240;
14  BRCODE  =  128;
15  DVCODE  =  22;
16  CLGTSAPPC =  193;
17  CLGTSAPPL =  1;
18  CLDTSTAPP =  194;
19  CLDTSTAPL =  1;
20  TPSUSZPC =  192;
21  TPSUSZPL =  1;
22  THFC    =  137;
23  THPL    =  127;
24  TRDAPC   =  136;
25  TRDLPL   =  83;
26  NREOT   =  99;

27  type
28    std = record
29       fixp : packed array [1..FIXPLEN] of char;
30       othl : packed array [1..OTHLEN] of char;
31    end;
32
33    std = "std"
34
35    contr = array[0..7] of integer;
36
37    ddd = record
38       ok : array[0..7] of integer;
39    end;
40
41    ddd = "ddd"
wdbb = record (window block format)
  err: packed array[1..2] of char
  vbadd: pstd
  siz: integer
 repid: integer
  off: integer
  len: integer
  wstats: integer
  sbuff: pdd
end

tskk = record
  nmr: array[1..2] of integer
end

efnn: integer

idss = integer

request = packed array[1..REDLEN] of char

wndnuo = 1..NODTRI
var
  a : integer;
  id : integer;
  j : integer;
  len : integer;
  rwndno : std;
  swnundo : std;
  userXperm : request;
  valid : boolean;
  wndno : wndno;

procedure buffclear(efn : efnn); extern;
procedure buffso(efn : efnn); extern;
procedure buffwait(efn : efnn); extern;
procedure fillb(b : integer);
  var c : char) extern;
procedure fillw(b : integer);
  var c1 : char
  var c2 : char) extern;
procedure initbuf(var swnundo : std); extern;
procedure readers(var userXreq : request;
  var len : integer); extern;
procedure readData(var wr : std); extern;
procedure tint(var id : integer); extern;
procedure tsend(swnundo : std);
  var wndno : wndno;
  id : integer); extern;

begin
  (* TERMA *)
  (* for display purposes on VT100 compatible terminals *)
  writeln(tty, chr(27), '[2J',
    chr(27), '[010F', chr(27), '[0a',

chr(27) + '[7124r' + chr(27) + '[?m' + chr(27) + '[A'];

writeln(Ltw, '##########################################################');
writeln(Ltw, '$');
writeln(Ltw, '$');
writeln(Ltw, 'TA'SK TERMA activated $');
writeln(Ltw, '$');
writeln(Ltw, '##########################################################');
writeln(Ltw, 'chr(27) + '[0m'));

wndno := 1;
($ initialize this to anything in valid range $)

initid();
($ create a dynamic region with name 'Al' $)

while (1 = 1) do ($ DO forever $)

begin

($ initialize send buffer $)

initbuf(wndno);

valid := true;
($ read type of user primitive $)

writeln(Ltw, 'readin users primitive type');

readterm(userXprim, len);
($ pad with blanks if needed $)

for j := (len + 1) to REQLEN do

userXprim[j] := '

if userXprim = 'tcro' then

begin
($ go in the tcreo file for input $)

reset(input, 'TCREO.TXT');

if ioread(input) < 0 then

begin

writeln(Ltw, 'Error opening the file called TCREO.TXT');

halt

end;

($ fill up the buffer with the appropriate fields $)

fillb(CRCDATE, wndno, fixp[i1] )

writeln(Ltw, 'enter the DESTINATION address');

readin(input); (($ skip over comment $)

readin(input, #)

fillw(snow, fixp[33] ; wndno, fixp[43] )

writeln(Ltw, 'enter the SOURCE address');
readln(input)); (* skip over comment *)
readln(input, a);
fillb(swindn, fixp[5], swindn, fixp[6]);
(* writeinttly('enter the CLASS and OPTION'))
readln(input)); (* skip over comment *)
readln(input, a);
fillb(swindn, fixp[8]);
(* writeinttly('enter the CALLING TSAP identifier'))
readln(input)); (* skip over comment *)
readln(input, a);
fillb(swindn, oth[4]);
fillb(CLGTSAFPC, swindn, oth[22]);
fillb(CLGTSAAPPL, swindn, oth[13]);
(* writeinttly('enter the CALLED TSAP identifier'))
readln(input)); (* skip over comment *)
readln(input, a);
fillb(swindn, oth[5]);
fillb(CLDTSAFPC, swindn, oth[3]);
fillb(CLDTSAAPPL, swindn, oth[6]);
(* writeinttly('enter the TPNU size'))
readln(input)); (* skip over comment *)
readln(input, a);
fillb(swindn, oth[10]);
fillb(TPUSZPL, swindn, oth[7]);
fillb(TPUSZPC, swindn, oth[8]);
(* writeinttly('enter the 4 THROUGHPUT values (8 octets)'))
readln(input)); (* skip over comment *)
read(input, a);
fillb(swindn, oth[11], swindn, oth[12]);
read(input, a);
fillb(swindn, oth[13], swindn, oth[14]);
read(input, a);
fillb(swindn, oth[15], swindn, oth[16]);
read(input, a);
fillb(swindn, oth[17], swindn, oth[18]);
(* writeinttly('enter the 4 TRANSIT DELAY values (8 octets)'))
readln(input)); (* skip over comment *)
read(input, a);
fillw(a, swinout.oth[27], swinout.oth[28])
read(input, a);
fillw(a, swinout.oth[29], swinout.oth[30])
read(input, a);
fillw(a, swinout.oth[31], swinout.oth[32])
readinput, a);
fillw(a, swinout.oth[33], swinout.oth[34])
fillb(TRDLPC, swinout.oth[225])
fillb(TRDLPL, swinout.oth[224])
else if userchar = 'tcrre' then
begin
(* do in the tcrres file for input *)
reset(input, 'TCRES.TXT')
if input < 0 then
begin
writeln('Error opening the file called TCRES.TXT ')
halt
end
(* fill up the buffer with the appropriate fields *)
fillb(CCCQDE, swinout.fox[1])
write(input, '"enter the SOURCE address"')
readin(input) (* skip over comment *)
readin(input, a);
fillw(a, swinout.fox[3], swinout.fox[6])
write(input, '"enter the CLASS and OPTION")
readin(input) (* skip over comment *)
readin(input, a);
fillw(a, swinout.fox[8])
write(input, '"enter the CALLING TSAP identifier")
readin(input) (* skip over comment *)
readin(input, a);
fillw(a, swinout.oth[4])
fillb(CLDTAPPCC, swinout.oth[22])
fillb(CLDTAPPL, swinout.oth[12])
write(input, '"enter the CALLED TSAP identifier")
readin(input) (* skip over comment *)
readin(input, a);
fillw(a, swinout.oth[5])
fillb(CLDTAPPCC, swinout.oth[33])
fillb(CLDTAPPL, swinout.oth[26])
241 ( writeln('enter the TPDU size');)
242 readln(input); (* skip over comment *)
243 readln(input, a);
244 fillb(10, swndo.oth[10]);
245 fillb(TPDUSZPL, swndo.oth[7]);
246 fillb(TPDUSZPL, swndo.oth[8]);
247
248 ( writeln('enter the 4 THROUGHPUT values (8 octets)');)
249 readln(input); (* skip over comment *)
250 read(input, a);
251 fillw(11, swndo.oth[11]); fillw(12, swndo.oth[12]);
252
253 fillw(13, swndo.oth[13]); fillw(14, swndo.oth[14]);
254 read(input, a);
255 fillw(15, swndo.oth[15]); fillw(16, swndo.oth[16]);
256 read(input, a);
257 fillw(17, swndo.oth[17]); fillw(18, swndo.oth[18]);
258
259 fillb(TRPLC, swndo.oth[9]);
260 fillb(TRPLC, swndo.oth[23]);
261
262 ( writeln('enter the 4 TRANSIT DELAY values (8 octets)');)
263 readln(input); (* skip over comment *)
264 read(input, a);
265 fillw(27, swndo.oth[27]); fillw(28, swndo.oth[28]);
266
267 fillw(29, swndo.oth[29]); fillw(30, swndo.oth[30]);
268 read(input, a);
269 fillw(31, swndo.oth[31]); fillw(32, swndo.oth[32]);
270 read(input, a);
271 fillw(33, swndo.oth[33]); fillw(34, swndo.oth[34]);
272
273 fillb(TRDPLC, swndo.oth[25]);
274 fillb(TRDPLC, swndo.oth[24]);
275 end (* trecs *)
276
277 else if userXe = 'tnadr' then (* ttnadr *)
278 begin (* fill up the buffer with the appropriate fields *)
279 fillb(DTCODE, swndo.fixe[11]);
280 fillb(NREDT, swndo.fixe[41]);
281 writeln('enter USER DATA');
282 readUdata(swndo)
283 end (* ttnadr *)
else if userXprim = 'tdreu' then (* tdreu *)
begin
  (* do in the tdreu file for input *)
  resul(input, 'TDREQ.TXT');
  if ioreusl(input) < 0 then
  begin
    writeln(tty, 'Error opening the file called TDREQ.TXT');
    halt
  end;
end;

(* fill up the buffer with the appropriate fields *)
fillb(DRCODE, swndno, fixp[1]);
write(tty, 'enter the DESTINATION address');
readln(input); (* skip over comment *)
readln(input, a);
fill(a, swndno, fixp[3], swndno, fixp[4]);
write(tty, 'enter the SOURCE address');
readln(input); (* skip over comment *)
readln(input, a);
fill(a, swndno, fixp[5], swndno, fixp[6]);
write(tty, 'enter the REASON for disconnection');
readln(input); (* skip over comment *)
readln(input, a);
fill(a, swndno, fixp[8]);
write(tty, 'enter the additional information for disconnection');
fillb(DVCODE, swndno,uth[2]);
readData(swndno);

(* tdreu *)
else
begin
  writeln(tty, 'error in inputed primitive');
  valid := false
end
if valid then
begin
  writeln(tty, 'USER A primitive successfully sent to TPRENTA');
  swnd(swndno, wndno, id) (* send buffer to TPRENTA *)
end;

(* DO forever *)
write(tty, 'Task TERMA terminated');
end.
NAME: TINIT

FUNCTION: This module is used to create a dynamic region for task TERTIA by issuing a CRRG$ memory management directive.

USAGE: procedure tinit(var id: integer); external

    tinit(id);

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: id - the region id.

MODULES CALLED: (The symbol '>' designates a primitive)

    >halt
    >writeln
    crrg

EXIT CONDITIONS: If there is an error detected in calling CRRG$, then an error message identifying the error and error code is printed onto standard I/O and the processor is then halted by using the 'halt' primitive.

The return code of the CRRG$ routine is set to greater or equal to 0 (zero) in case of successful completion and to less than 0 in case of unsuccessful completion.

For more information on the use of CRRG$ and its return codes, see pages 6-36, 6-38 of the RSX-11M V3.2 Executive Reference Manual.

REMARKS: At the moment, there is no protection on the region. One can easily remedy this by giving an appropriate value to the region protection word.
1  const
2     REDLEN  =  6;
3     NOPTR   =  7;
4     NWDSZ   =  3;
5     APRNO   =  7;
6     REGSZ   =  21;
7     FIXLEN  =  8;
8     OTHLEN  = 130;
9     STRLEN  = 118;
10    CRCODE  = 224;
11    CCCODE  = 208;
12    DTDCODE = 240;
13    DRCODE  = 128;
14    DUCODE  = 224;
15    CLGTSAPPC = 193;
16    CLGTSAPPL =  1;
17    CLDTSAPPC = 194;
18    CLDTSAPPL =  1;
19    TPDUSZPC = 192;
20    TPDUSZPL =  1;
21    THPC    = 137;
22    TMPL    = 121;
23    TRDLPZ    = 136;
24    TRDLP    =  8;
25    NREOT    = 99;
26
27  type
28     std = record
29        fixp : packed array [1..FIXLEN] of char;
30        uth : packed array [1..OTHLEN] of char;
31     end;
32
33  pstd = ^std;
34
35  ctrl1 = array[0..7] of integer;
36
37  ddd = record
38        ok : array[0..7] of integer;
39     end;
40
41  pdd = ^ddd;
44  wdbb = record (* window block format *)
45     wnr : packed array[1..2] of char
46     vbbdd : rstd;
47     siz : integer;
48     resid : integer;
49     ofss : integer;
50     len : integer;
51     wsata : integer;
52     srbuff : rudi;
53     end;
54
55  tskk = record
56     nwm : array[1..2] of integer;
57     end;
58
59  efnn = integer;
60
61  idss = integer;
62
63  request = packed array[1..REOLEN] of char;
64
65  wnnnoo - 1..NOFTR;}
procedure tinit(var id : integer);
  var
    control : control;
    ids    : idss;
  procedure crrs(var control : control; var ids : idss); extern (fortran);
begin
  writeln(tty, 'entering tinit subroutine');
  (* create and attach to region with region name 'AI' *)
  (* control[0] region id returned *)
  control[1] := REG52; (* size of the region in 32-word blocks also returned *)
  control[2] := 2840; (* region name (2 words in Radix-30 format) *)
  control[3] := 0; (* or 0 for no-name, 'AI' *)
  control[4] := 11414; (* name of the partition that contains the *)
  control[5] := 0; (* region. (2 word in Radix-30 format *)
                 (* presently it is D6N *)
                 (* or 0 for the partition in which the task is *)
                 (* running *)
  control[6] := 35; (* Region status word (see page 3-14) also returned *)
                 (* bits set: *)
                 RS.ATT := 32
                 RS.WRT := 2
                 RS.RED := 1
  control[7] := 0; (* Region protection code *)
  (* everything is permitted see > 3-13 *)
  crrs(control, ids);
  writeln(tty, 'id assigned to the created region = ', control[0]);
  writeln(tty, 'size in 32-W blocks of the attached region = ', control[1]);
  writeln(tty, 'region status word = ', control[6]);
  writeln(tty, 'ids = ', ids);
  if ids < 0 then
    begin
      writeln(tty, 'error on crrs error code = ', ids);
      halt
    end;
  id := control[0] (* stick region id in appropriate return field *)
91   { writeln('leaving tinit subroutine'); } 
92   end.

NO ERROR DETECTED
TOTAL PROGRAM SIZE   000450
OUTERMOST DATA SIZE  000002
RESERVED STACK & HEAP 000542
COMPILATION TIME 13 SECONDS

PAS TINIT,TINIT,P42-TINII
NAME:     TPRENTA (MAIN)

FUNCTION: To serve as Transport Protocol Entity for USER "A".

USAGE:    from MCR: "RUN TPRENTA TASK=TPRENTA INC.100."

INPUT PARAMETERS: NONE

OUTPUT PARAMETERS: NONE

MODULES CALLED:  (The symbol "\"" designates a primitive)
chr
new
ord
write
writeln
buffclear
buffso
buffwait
initial
интлду
предду
река
сендд
семра
семра

EXIT CONDITIONS: This program has been designed to run indefinitely until
the task into which it runs is specifically aborted with
the MCR command "ABORT TPRENTA".

If an error is detected by one of the routines called,
then the routine itself will print an error message and
halt the processor by using the "halt" primitive.

REMARKS: The format of the TPUs is different from the standard
format since the PDP-11 stores the Hi and LO bytes of a
word in memory. So internally speaking (in memory) the
44 IF IOWs are according to standards.
45
46 **************************************************
47
48 Program TPRENTA(tty);
49
50 ($I+TPRENTA.DEF)
const

FIXPLEN    =  8;
OTHLEN    = 130;
STRLEN    = 118;
METROLEN  =  6;
CRCODE    = 224;
CCCODE    = 208;
DTCODE    = 240;
DRCODE    = 128;
NDROCODE  = 120;
ERCODE    = 112;
APRNO     =  7;
WNDSZ     =  3;
REGSZ     =  21;  \{ NODTR 'times WNDSZ \}
NOPTR     =  7;
DUHLEN    = 101;
DUMMLEN   =  5;
NODATSLG  =  20;
ICCODE    = 193;
DVCODE    = 224;
ERRLEN    = 128;
GCDEST    = 2222;
GCSOUR    = 1111;
TPDUSZPL  =  11;
TPDUSZPC  = 192;
THPC      = 137;
GCTPDU    = 128;
GCTHR1    =  1;
GCTHR2    =  2;
GCTHR3    =  3;
GCTHR4    =  4;
THPL      = 121;
TRDLPL    =  81;
TRDLPC    = 136;
GCTR1     =  71;
GCTR2     =  81;
GCTR3     =  91;
GCTR4     = 101;
NREOT     =  99;
GCRESO    = 111;
UNDEF     =  99;
type

std - record

fixp : packed array [1..FIXPLEN] of char;

uth : packed array [1..OTHLEN] of char;
end;

pstd = 'std';

ctrl = array[0..7] of integer; ( format for CRGB$)

netXreq = packed array[1..NETXREQLEN] of char;

dd = record

req : netXreq;

dum : array[1..DUMLEN] of integer;
end;

rdd = 'rdd';

tskk = record

nam : array [1..2] of integer;
end;

rbuff = record ( format for RCVD$)

nam : tskk;

req : netXreq;

dum : array [1..DUMLEN] of integer;
end;

buff = record ( format for SDAT$)

req : netXreq;

dum : array [1..DUMLEN] of integer;
end;

rddd = record

uk : array[1..2] of integer;

req : netXreq;

dum : array[1..DUMLEN] of integer;
end;

rddd = 'rddd';
rwdbb = record ( format for CRAW$ and RREF$ )
  rarr : packed array[1..2] of char;
  rvbadd : psdt;
  rsiz : integer;
  rresid : integer;
  roffs : integer;
  rlen : integer;
  rwstats : integer;
  rsrbuff : rwddi
end;

wdbb = record ( format for CRAW$ and SREF$ )
  warr : packed array[1..2] of char;
  wvbadd : wstd;
  wsiz : integer;
  wresid : integer;
  woffs : integer;
  wlen : integer;
  wwstats : integer;
  wsrbuff : wddi
end;

state = (CLOSED, OPEN, WFCC, WFTRESP, WNFNC); (states of TPRENTA)

orXsts = (open, closed, inprogress);

waessXsts = (zerorone,twothree,four);

waessXerr = (fivesixseven,eighth,nineteen,eleven,
twelve,thirteen,fourteen,fifteen,sixteen,seventeen,eighteen,
nineteen,twentysix,seventyone);

idss = integer;
efnn = integer;
tagg = integer;
tnlt = integer;
wndnoo = 1..NPTIRI;

arr = packed array[1..2] of char
(* FORMAT OF TPDUs  [record of type 'std']

Connect request:
fixp[1]  - CRCDI
fixp[2]  - LI
fixp[3]  - not used
fixp[4]  - not used
fixp[5]  - SRCREF (LO)
fixp[6]  - SRCREF (HI)
fixp[7]  - not used
fixp[8]  - CLASSXDP

oth[1]  - CLGTSAPPL
oth[2]  - CLGTSAPPC
oth[3]  - CLDTSAPPC
oth[4]  - CLGTSAPVLU
oth[5]  - CLDTSAPVLU
oth[6]  - CLDTSAPPL
oth[7]  - TPDUSZPL
oth[8]  - TPDUSZPC
oth[9]  - THPC
oth[10] - TPDUSZVL
oth[11..22] - THVL
oth[13..14] - THROUGHPUT 1
oth[15..16] - THROUGHPUT 2
oth[17..18] - THROUGHPUT 3
oth[19..22] - not used

oth[23]  - THPL
oth[24]  - TRDLPL
oth[25]  - TRDLPC
oth[26]  - not used
oth[27..34] - TRDLVL
oth[35]  - not used
oth[36..130] - not used

Connect confirm:
fixp[1]  - CCCDI
fixp[2]  - LI
fixp[3]  - not used
fixp[4]  - not used
fixp[5]  - SRCREF (LO)
PASCAL PDP-11 VERSION 6.07 1908-08-04 04:30:17 PAGE 8

TPRENTA,PAS

95  fixr[6] - SRCREF (H1)
96  fixr[7] - not used
97  fixr[8] - CLASXOP
98
99  oth[1] - CLGTSA_PLL
100  oth[2] - CLGTSA_PLL
101  oth[3] - CLDTSA_PLL
102  oth[4] - CLGTSA_PLL
103  oth[5] - CLDTSA_PLL
104  oth[6] - CLDTSA_PLL
105  oth[7] - TPDU2ZPL
106  oth[8] - TPDU2ZPL
107  oth[9] - THPC
108  oth[10] - TPDU2ZPL
109  oth[11..22] - THVL
110  oth[13..14] - THROUGHPUT 2
111  oth[15..16] - THROUGHPUT 3
112  oth[17..18] - THROUGHPUT 4
113  oth[19..22] - not used

114  oth[23] - THPL
115  oth[24] - TRDLFL
116  oth[25] - TRDLPF
117  oth[26] - not used
118  oth[27..34] - TRDLVL
119  oth[35] - not used
120  oth[36..130] - not used

121  Data transfer:
122  fixr[1] - DTC
123  fixr[2] - LI
124  fixr[3] - not used
125  fixr[4] - NREG
126  fixr[5..8] - not used
127  oth[11..120] - USER DATA
129  oth[12] - not used
130  oth[13..120] - string
131  oth[121..130] - not used

132  Error:
133  fixr[1] - ERC
134  fixr[2] - LI
135  fixr[3] - DSTREF (LD)
 disconnect request:
 fixr[1] - drc
 fixr[4] - dstref (hi)
 fixr[5] - srcrff (lo)
 fixr[6] - srcrref (hi)
 fixr[7] - not used
 fixr[8] - reason

 oth[1] - dul
 oth[2] - duc
 oth[3..120] - dvvll
 oth[121..130] - not used

 var
dstref : arri
err : arri
id : integer
messxterm : boolean
netxcon : boolean
mxreq : mxreq
openxsts : opxsts
ptrwnd : stdi
queue : boolean
reason : integer
rjcse : integer
srcrref : arri
stdxprenta : state
valid : boolean
wndno : wndno
procedure buffclear(efn : efn); external
procedure buffs0(efn : efn); external
procedure buffwait(efn : efn); external
procedure initial(var id : integer); external
procedure insptdu(ptrwnd : pstd);
    var valid : boolean;
    var rcse : integer;
    var err : errr; external
procedure prepptdu(ptrwnd : pstd);
    var dstref : errr
    var srcref : arrri
    rcse : integer;
    err : errr
    cod : integer); external
procedure recpt(var ptrwnd : pstd);
    var nXreq : netXrea
    var queue : boolean;
    var messXterm : boolean); external
procedure sendd(ptrwnd : pstd);
    var window : window;
    nXreq : netXrea
    id : integer;
    bol : boolean); external
procedure werra(ptrwnd : pstd);
    rcse : integer;
    aes : wmassXerr); external
procedure wterma(ptrwnd : pstd);
    aes : wmassXsts); external

begin (* MAIN *)

(* for display purposes on VT100 terminal *)
' writeln(tty, chr(27), '(',2');
225   chr(27), '(',0100', chr(27), '(',0m';
226   chr(27), '(',7124', chr(27), '(',0m, chr(27), '(',A'');
227   writeln(tty, '****************************');
228   writeln(tty, '#');
229   writeln(tty, ' TASK TPRENTA activated' );
230   writeln(tty, '#');
231   writeln(tty, '****************************');
232   writeln(tty, chr(27), '(',0m');
233
234   wndno := 1;  (* initialize sending window *)
235   rvse := 0;   (* initialize reception cause *)
236   stXrenta := CLOSED;  (* initialize state of TPRENTA *)
237   openXsts := closed;  (* initialize open status of TPRENTA *)
238   netXcon := false;  (* initialize status of network connection *)
239   new('strand);  (* create dynamic space for TPOD *)
240   initial(id);  (* create a dynamic region with a name of 'AC' *)
241
242   (* make sure the screen is clear for display purposes *)
243   writeln(tty);writeln(tty);writeln(tty);writeln(tty);writeln(tty);writeln(tty);writeln(tty);
244   writeln(tty);writeln(tty);writeln(tty);writeln(tty);writeln(tty);writeln(tty);writeln(tty);
245   writeln(tty);writeln(tty);writeln(tty);writeln(tty);writeln(tty);writeln(tty);writeln(tty);
246   writeln(tty);writeln(tty);writeln(tty);writeln(tty);writeln(tty);writeln(tty);writeln(tty);
247   writeln(tty);writeln(tty);writeln(tty);writeln(tty);writeln(tty);writeln(tty);writeln(tty);
248   writeln(tty);writeln(tty);writeln(tty);writeln(tty);writeln(tty);writeln(tty);writeln(tty);
249  while(l = 1) do  (* DO this forever *)
250      begin
251
252      (* print out the state of the transport protocol entity *)
253      writeln(tty, chr(27), '(',7120', 'STATE < '/');
254      if (ord(stXrenta) = 0) then
255         writeln(tty, 'CLOSED ');
256      else if (ord(stXrenta) = 1) then
257         writeln(tty, 'OPEN ');
258      else if (ord(stXrenta) = 2) then
259         writeln(tty, 'WFC ');
260      else if (ord(stXrenta) = 3) then
261         writeln(tty, 'WFR ');
262      else
263        writeln(tty, 'WFR ');
264
265      (* receive packet from receive by reference queue or by task queue *)
266      recu('strand); received queue messXrenta);
267 \# print out the primitive and the type of TPDU received
268 if messXterm then
269 begin
270   writeln(tty, chr(27), '\033[120f', 'RECEIVED: ', nXrea);
271   writeln(tty, chr(27), '\033[120f', 'TPDU: NONE
272 end
273 else if messXterm then
274 begin
275   writeln(tty, chr(27), '\033[120f', 'RECEIVED: ')
276   if ord(strwnd).fixp[1] = CRCODE then
277     writeln(tty, 'CRC
278   else if ord(strwnd).fixp[1] = CCCODE then
279     writeln(tty, 'CCC
280   else if ord(strwnd).fixp[1] = DTCODE then
281     writeln(tty, 'DT
282   else if ord(strwnd).fixp[1] = DRCODE then
283     writeln(tty, 'DRC
284   else
285     writeln(tty, 'XX
286   writeln(tty, chr(27), '\033[120f', 'TPDU: NONE
287 end
288 else
289 begin
290   writeln(tty, chr(27), '\033[120f', 'RECEIVED: ')
291   if ord(strwnd).fixp[1] = CRCODE then
292     writeln(tty, 'CR
293   else if ord(strwnd).fixp[1] = CCCODE then
294     writeln(tty, 'CC
295   else if ord(strwnd).fixp[1] = DTCODE then
296     writeln(tty, 'DT
297   else if ord(strwnd).fixp[1] = DRCODE then
298     writeln(tty, 'DR
299   else if ord(strwnd).fixp[1] = ERCODE then
300     writeln(tty, 'ER
301   else
302     writeln(tty, 'XX
303 end;
304 writeln(tty, chr(27), '\033[120f', '\
305 if messXterm then
306 begin
(* message from term ? *)
case stXprenta of
   CLOSED:
      begin
         if (ptrwnd^.fix[1] = chr(CRCODE)) then
            begin
               if (not(netXcon)) then
                  begin
                     begin
                        openXsts := inprogress;
                        preptedu(ptrwnd, dstref, srcref, rjces, err, CRCODE);
                        sendd(ptrwnd, wndno, 'ncrea', id, true);
                        stXprenta := WFNC
                     end
                  end
               else
                  begin
                     if ((netXcon) and (openXsts = inprogress)) then
                        (* P4 *)
                     stXprenta := WFNC
                  end
               begin
                  if ((netXcon) and (openXsts = opens)) then
                     (* P3 *)
                  begin
                     stXprenta := WFCC;
                     preptedu(ptrwnd, dstref, srcref, rjces, err, CRCODE);
                     sendd(ptrwnd, wndno, nXrea, id, false);
                     (* P3 *)
                     else
                        werra(ptrwnd, rjces, five)
                        (* true unacceptable P0 *)
                     end
                  end
               end
            end
         end
      begin
         if (ptrwnd^.fix[1] = chr(DTCODE)) then
            begin
               preptedu(ptrwnd, dstref, srcref, rjces, err, DT CODE);
               sendd(ptrwnd, wndno, nXrea, id, false);
               (* send to NTRNSDA *)
            end
      else
         begin
            werra(ptrwnd, rjces, six)
            (* closed state *)
         end;
begin
  if (ptrwnd^.fixp[1] = chr(DRCODE)) then begin
    netXcon := false;
    openXsts := closed;
    sendd(ptrwnd, window, 'ndrea ', id, true); (* send nxrea to NTRNSDA *)
    stXrenta := CLOSED;
  end
else
  weira(ptrwnd, rcse, seven);
end (* OPEN state *)

WFCC:
begin
  if (ptrwnd^.fixp[1] = chr(DRCODE)) then begin
    netXcon := false;
    openXsts := closed;
    sendd(ptrwnd, window, 'ndrea ', id, true); (* send nxrea to NTRNSDA *)
    stXrenta := CLOSED
  end
else
  wierra(ptrwnd, rcse, eight) (* WFCC state *)
end

WTFRESP:
begin
  if (ptrwnd^.fixp[1] = chr(CCCODE)) then begin
    pprentdu(ptrwnd, dstref, srcRef, rcse, err, CCCODE);
    sendd(ptrwnd, window, nxrea, id, false); (* send to NTRNSDA *)
    stXrenta := OPEN
  end
else begin
  (* orres *)
end
begin
  if (ptrwnd^.fixp[1] = chr(DRCODE)) then begin
    pprentdu(ptrwnd, dstref, srcRef, rcse, err, DRCODE);
    sendd(ptrwnd, window, nxrea, id, false); (* send to NTRNSDA *)
    stXrenta := CLOSED
  end
else
werrp(ptrwnd, rjuse, nine)
end
end

WFNC:
begin
  if (ptrwnd$.fix[1] = chr(DRCODE)) then
    begin
      netXcon := false;
      openXcts := closed;
      sendd(ptrwnd, wndnor, 'ndrej', id, true);
      stXprenta := CLOSED
    end
  else
    begin
      werrp(ptrwnd, rjuse, ten);
    end
end

begin
  case stXprenta of
    CLOSED:
      begin
        if (queue) then
          begin
            if (nXrea = 'ncind') then
              begin
                netXcon := true;
                openXcts := open;
                sendd(ptrwnd, wndnor, 'ncres', id, true)
              end
            else if (nXrea = 'ndind') then
              begin
                openXcts := closed;
                netXcon := false
              end
            end
        end
  end
if (strwnd^.fix[1] = chr(CRCode)) then
begin
  insertru(strwnd, valid, rcsre; err);
  if valid then
  begin
    sxprenta := WFTRESP;
    dstref[1] := strwnd^.fix[5];
    dstref[2] := strwnd^.fix[6];
    wterma(strwnd, one)
  end
  else
  begin
    werrr(strwnd, rjses, nineteen);
    repflp(strwnd, dstref, srcref, rjses, err, NDRCODE);
    sendd(strwnd, wndno, nxrea, id, false)
  end
end
end;

OPEN:
begin
  if (queue) then
  begin
    if (nxrea = 'wind') then
    begin
      wterma(strwnd, zero);
      netxcon := false;
      openxsts := closed;
      sendd(strwnd, wndno, 'ndrea', id, true); (* give TDIND to the TS over A *)
      (* [1] *)
      if (not netxcon) then
      sendd(strwnd, wndno, 'nrres', id, true); (* send nxrea to HTRNSDA *)
      (* [5] *)
      stxprenta := CLOSED
    end
  else
  begin
    if (nxrea = 'mind') then
    begin
      (* [ind] *)
      end
end
end
end;
482  netXcon := false;
483  openXsts := closed;
484  uterma(ptrwnd, zero);
485  stXprenta := CLOSED;
486  end
487  end
488  end
489  else
490  begin
491    if (ptrwnd^.fixP[1] = chr(DTCODE)) then
492      begin
493        instedu(ptrwnd, valid, rcuser, err);
494        if valid then
495          uterma(ptrwnd, three)  (* give THORD to user R *)
496        else
497          werr(ptrwnd, rcuser, sevenlen);
498          pretrd(ptrwnd, data, srcerr, rcuser, err, ERCODE);
499          sendd(ptrwnd, wndnor, nXrea, id, false);  (* send ER TO UTRNSDA *)
500      end
501    end
502    end
503    else
504    begin
505      if (ptrwnd^.fixP[1] = chr(ERCODE)) then
506        begin
507          netXcon := false;
508          openXsts := closed;
509          sendd(ptrwnd, wndnor, nXrea, id, true);  (* send nXrea TO UTRNSDA *)
510          stXprenta := CLOSED
511        end
512    end
513    else
514    begin
515      if (ptrwnd^.fixP[1] = chr(ERCODE)) then
516        begin
517          werr(ptrwnd, rcuser, eleven);
518          netXcon := false;
519          openXsts := closed;
520          sendd(ptrwnd, wndnor, nXrea, id, true);  (* send nXrea TO UTRNSDA *)
521          stXprenta := CLOSED
522        end
523    end
524    end
end

WFNC :
begin
if (queue) then
begin
if (nXrea = 'nccon') then
begin
netXcon := True;
openXsts := 'true';
sendd(ptrwnd, wndno, nXrea, id, false);
stXrenca := WFCC
end
else
begin
if (nXrea = 'nwind') then
begin
wterm(ptrwnd, zero);
netXcon := False;
openXsts := 'closed';
sendd(ptrwnd, wndno, 'nwind', id, true);
if (not netXcon) then
sendd(ptrwnd, wndno, nXrea, id, true);
stXrenca := CLOSED
end
else
begin
if (nXrea = 'ndind') then
begin
netXcon := False;
openXsts := 'closed';
wterm(ptrwnd, zero);
stXrenca := CLOSED
end
end
end
end
else
begin
wterm(ptrwnd, njssel, fifteen)
end;
WFCC:
    begin
    if (queue) then
        begin
            if (nXrea = 'nind') then
                begin
                    wterma(ptrwnd, zero);
                    netXcon := false;
                    openXsts := closedst;
                    sendd(ptrwnd, wndno, 'ndrea', id, true);
                end
            if (not netXcon) then
                sendd(ptrwnd, wndno, 'nres', id, true);
            end
            stXrenta := CLOSED
        end
    else
        begin
            if (nXrea = 'mind') then
                begin
                    wterma(ptrwnd, zero);
                    netXcon := false;
                    openXsts := closedst;
                    stXrenta := CLOSED
                end
        end
    end
    end
else
    begin
    if (nXrea = 'queue') then
        ("\# if queue")
    end
else
    begin
    if (ptrwnd^.fixp[1] = chr(CCCODE)) then
        begin
            inxtrdu(ptrwnd, valid, rjose, err);
        end
    if valid then
        begin
            wterma(ptrwnd, two);
            stXrenta := OPEN
        end
    else
        begin
            werra(ptrwnd, msgc, twelve);
            netXcon := false;
            openXsts := closedst;
            sendd(ptrwnd, wndno, 'ndrea', id, true);
        end
    end
end
611  stXrenta := CLOSED
612  end
613  end
614  else
615  begin
616  if (Artwmd^.fix[t] = chr(ERRCODE)) then
617  begin
618  werr(artwmd, lenor, thirteen);  (* with reasons etc.*)
619  netXcon := false;
620  openXsts := closed;
621  sendd(artwmd, wndnor, 'ndrea', idr, true);  (* send nXreo to NTRNSDA *)
622  stXrenta := CLOSED
623  end;  (* er *)
624  else
625  begin
626  if (Artwmd^.fix[t] = chr(ERRCODE)) then
627  begin
628  instrd(artwmd, valid, lenor, err);
629  if valid then
630  begin
631  wterm(artwmd, four);  (* give TDIND to user A *)
632  netXcon := false;
633  openXsts := closed;
634  sendd(artwmd, wndnor, 'ndrea', idr, true);  (* send nXreo to NTRNSDA *)
635  stXrenta := CLOSED
636  end
637  else
638  begin
639  werr(artwmd, lenor, eighteen);
640  prestpd(artwmd, dstref, srcref, rscer, err, ERRCODE);
641  sendd(artwmd, wndnor, nXreo, idr, false).  (* send ER TPDU to NTRNSDA *)
642  end
643  end
644  end
645  end
646  end
647  end
648  end
649  WTRRESP :=
650  begin
651  if (queue) then
652  begin
653  if (nXreo = 'nwnd') then
begin
  writem(ptrwnd, 'zero');
  netXcon := false;
  openXsts := closed;
  sendd(ptrwnd, window 'nXrea ', id, true);
  if (not netXcon) then
    sendd(ptrwnd, window 'nXrea ', id, true);
  stxrenta := CLOSED
end
else
  begin
    if (nXrea = 'ndind ') then
      begin
        netXcon := false;
        openXsts := closed;
        writem(ptrwnd, 'zero');
        stxrenta := CLOSED
      end
    end
    else
      writem(ptrwnd, 'nXrea ', sixteen)
end
end;

writeln(tty, 'TPRENTA TASK TERMINATED........................

NO ERROR DETECTED
TOTAL PROGRAM SIZE 020710
OUTERMOST DATA SIZE 000044
RESERVED STACK & HEAP 001072
COMPILATION TIME 217 SECONDS
PAS TPRENTA, TPRENTA/P42=TPRENTA
NAME: TSEND

FUNCTION: To send a buffer of information to task IFRENTA, it uses the
send by reference directive SREF$.

USAGE: procedure tsend(swindno: stdi;
var wndno: wndno;
    id: integer) external

sendd(swindno: wndno, wndno, id);

INPUT PARAMETERS:
    swindno - the buffer to send by reference.
    wndno - the window number to be used(1 to 7),
    id - the region id.

OUTPUT PARAMETERS:
    wndno - the new window number.

MODULES CALLED:
(The symbol '*' designates a Primitive)

>halt
> writeln
    crw
    sref

EXIT CONDITIONS:
If there are any errors detected in calling CRAW$ and
SREF$, then an error message identifying the error
and error code is printed onto standard I/O and the
processor is then halted by using the 'halt' Primitive.
The return code of both these system directives is set
to greater or equal to 0(zero); in case of successful
completion and to less than 0 in case of unsuccessful
completion.

For more information on the use of CRAW$ and SREF$ and
their return codes, see pages 6-31, 6-34 and 6-138, 6-140
REMARKS:
The method used in this module involves putting information in the return windows and sending these by reference.
Each call to this routine involves sending the next window in line (i.e., the next window in the return or window number one if the last window used was used on the previous call). This strategy guarantees that at least 7 'SENDBS' can be done before one of the data sets overwrites the new data.

*******************************************************************************

(*SEMMA.DEF)
1  const
2     REOLEN = 6;
3     NOPTR  = 7;
4     WNGSIZ = 13;
5     APRNO  = 71;
6     REGSZ  = 211;
7     FIXPLEN = 8;
8     OTHLEN = 130;
9     STRLEN = 118;
10    CRCODE = 224;
11    CCCODE = 206;
12    DTCODE = 240;
13    DRCODE = 128;
14    DVCODE = 224;
15    CLGTSAPFC = 193;
16    CLGTSAPPL = 11;
17    CLDTSAAPFC = 194;
18    CLDTSAAPPL = 11;
19    TPDUSZPC = 192;
20    TPDUSZPL = 11;
21    THPC = 137;
22    THPL = 121;
23    TRDLPC = 136;
24    TRDLPL = 81;
25    NREOT = 99;
26
27  type
28     std = record
29         fixp : packed array [1..FIXPLEN] of char;
30         oth : packed array [1..OTHLEN] of char;
31     end;
32
33  fstd = std;
34
35  cont1 = array[0..7] of integer;
36
37  ddd = record
38         uk : array[0..7] of integer;
39     end;
40
41  pdd = ddd;
wdbb = record (window block format)
  enr : packed array[1..2] of char
  vbadd : stdi
  size : integer
  regid : integer
  offset : integer
  len : integer
  wstats : integer
  srbuf : stdi
end

lshk = record
  idaw : array[1..2] of integer
end

efmt = integer
idss = integer
request = packed array[1..REQLEN] of char
wndno = 1..NOPTRI
procedure tsend(swndnu: std); 
var wndnu: wndnuo;
  id: integer;

var 
  efn: efnn;
  ids: idss;
  isrb: ddd;
  swdb: wdbbi;
  tsk: tskki;
  wdb: wdbbi;

procedure crw(var wdb: wdbbi; var ids: idss); extern(fortran);

procedure sref(var tsk: tskki); 
var efn: efnn;
  var swdb: wdbbi;
  var isrb: ddd;
  var ids: idss); extern(fortran);

begin
  writeln(tty: 'entering tsend procedure');
  writeln(tty: 'window number is ', wndnu);
  (* set up task name * 'TPRENA' *)
  tsk.nam[1] := '32638';

  (* create a send window in the region *)
  wdb.wnr[2] := chr(APRND);    (* Active Page Register, *)
  wdb.siz := WNDSZ;            (* the size of the window in 32-word blocks *)
  wdb.resid := id;             (* the region id *)
  wdb.off := (wndnu-1)*WNDSZ;  (* the offset in region *)
  wdb.len := WNDSZ;            (* the length to win *)
  wdb.wstatus := 386;          (* window status word *)

  crw(wdb, ids);

  writeln(tty: 'window created ids = ', ids);
  writeln(tty: 'id assigned to the window = ', ord(wdb.wnr[1]));
  writeln(tty: 'virtual base address = ', ord(wdb.wadr));

102  writeln(tty, 'length actually mapped in 32-word blocks = ', wdb.len);
103  writeln(tty, 'window status word = ', wdb.wstats);
104  if ids < 0 then
105  begin
106  writeln(tty, 'error in using cram, error code = ', ids);
107  halt
108  end;
109
110  (* stick buffer into the send window *)
111  wdb.vbadd.f1 := swndw.f1;
112  wdb.vbadd.oth := swndw.oth;
113
114  (* prepare window definition block to send by reference *)
115  swdb.resid := id;
116  swdb.offs := (wndno - 1) * WNDSZ; (* offset in region (32W blocks) *)
117  swdb.len := WNDSZ; (* length to map (32W blocks) *)
118  swdb.wstats := 3;
119  (* receiver access permission same as sender *)
120  (* bits set: *)
121  WS.WRT = 2
122  WS.RED = 1
123
124  sref(lsk, ref, swdb, isrb, ids);
125  writeln(tty, 'ids for sref is = ', ids);
126  writeln(tty, 'code is = ', ord(wdb.vbadd.f1[1]));
127  if ids < 0 then
128  begin
129  writeln(tty, 'error in using sref, error code = ', ids);
130  halt
131  end;
132
133  (* change window for next call *)
134  if wndno = NOPTR then
135  wndno := 1
136  else
137  wndno := wndno + 1
138  writeln(tty, 'leaving tsend procedure');
139 end.

NU ERROR DETECTED
TOTAL PROGRAM SIZE 001326
OUTERMOST DATA SIZE 000002
RESERVED STACK & HEAP 0000610
COMPILATION TIME 23 SECONDS
NAME: VAL

FUNCTION: This function takes two characters c1(LO) and c2(HI) and returns the integer that the combination of HI-LO forms.

USAGE: i := integer;

function val(c1 : char; c2 : char) : integer; external;

i := val(c1, c2);

INPUT PARAMETERS: c1 - the LO character.
c2 - the HI character.

OUTPUT PARAMETERS: (returned) - the integer formed by HI-LO.

RULES CALLED: (The symbol '"' designates a primitive)

>halt
>writeln

EXIT CONDITIONS: Normal exit conditions (top-down)

REMARKS: One should be aware that the hi and low bytes of a word are positioned as LO-HI in memory on the PDP-11.

function val(c1 : char; c2 : char) : integer;

type tr1c = record
  (% variant part %)
case bool : boolean of
    false : ( w : integer );
    true : ( c : packed array [1..2] of char )
  end;


VAR tt : trick;
BEGIN
{ writeln(tty, 'entering val function') }

tt.bol := true;
{ put the characters in the variant record }
tt.c[1] := c1;
tt.c[2] := c2;
{ assign val the value of the character combination }
val := tt.w;
{ writeln(tty, 'leaving val subroutine') }

END.

NO ERROR DETECTED
TOTAL PROGRAM SIZE 000230
OUTERMOST DATA SIZE 000002
RESERVED STACK & HEAP 000524
COMPILATION TIME 5 SECONDS

PAS VAL, VAL/P42=VAL
IZED

NAME: WERRA

FUNCTION: Write an appropriate error message on User A's terminal due to an error situation in TPRENTA.

USAGE: procedure werra(ptrwnd : pstdipjrcjsepinter; mes : wmessXerr); extern;

werra(ptrwnd, pjcjsep, mes);

INPUT PARAMETERS: ptrwnd - pointer to received TPDU.
prjsep - cause for rejection of TPDU.
mes - the error message number.

OUTPUT PARAMETERS: NONE

MODULES CALLED: (The symbol '>' designates a primitive)
>ord
>writeIn

EXIT CONDITIONS: Normal exit conditions.

REMARKS: The TPRENTA module should be studied for a better understanding of the error encountered.

*******************************************************************************

(*$+TPRENTA.DEF)
1 2  const
3  FIXPLEN = 8;
4  OTHLEN = 130;
5  STRLEN = 118;
6  NETREQLEN = 6;
7  CRCODE = 224;
8  CCCODE = 208;
9  DTCODE = 240;
10  DRCODE = 128;
11  NDRCODE = 120;
12  ERCODE = 112;
13  APRNO = 7;
14  WNDSZ = 3;
15  REGSZ = 21;  { NOPTR times WNDsz }
16  NOPTR = 7;
17  DUHLEN = 10;
18  DUMMLLEN = 5;
19  NDATFLG = 20;
20  IVCODE = 193;
21  DVCODE = 224;
22  ERRLEN = 128;
23  GCDST = 2222;
24  GCSOUR = 1111;
25  TPDUSZPL = 1;
26  TPDUSZPC = 192;
27  THPC = 137;
28  GCTFDU = 128;
29  GCTHR1 = 11;
30  GCTHR2 = 21;
31  GCTHR3 = 31;
32  GCTHR4 = 41;
33  THPL = 121;
34  TDPL = 81;
35  TRDPLC = 136;
36  GCTR1 = 71;
37  GCTR2 = 81;
38  GCTR3 = 91;
39  GCTR4 = 101;
40  NRET = 991;
41  GCRESD = 111;
42  UNDEF = 991;
type
  std = record
    fixp: packed array [1..FIXPLEN] of char;
    oth: packed array [1..OTHLEN] of char;
  end;
  *std = std;

contrl = array[0..7] of integer; (format for CRDG$)

netXreq = packed array[1..NETREQLN] of char;

ddd = record
  rev : netXreq;
  dum : array[1..DUHLEN] of integer;
end;

edd = "ddd"

etsk = record
  nam : array [1..2] of integer;
end;

rbuff = record (format for RCVD$)
  nam : etsk;
  req : netXreq;
  dum : array [1..DUHLEN] of integer;
end;

buff = record (format for SDAT$)
  rev : netXreq;
  dum : array [1..DUMLEN] of integer;
end;

rddd = record
  uk : array[1..2] of integer;
  req : netXreq;
  dum : array[1..DUHLEN] of integer;
end;

rddd = "rddd"
rwdbb = record { format for CRAW$ and RREF$ }
  rsrc : packed array[1..2] of char;
  rvbadd : stdfl
  rsiz : integer;
  rresid : integer;
  roffs : integer;
  rlen : integer;
  rsstats : integer;
  rsbuff : rddfl
end;

rwdbb = record { format for CRAW$ and SREF$ }
  ssrc : packed array[1..2] of char;
  vbadd : stdfl
  size : integer;
  resid : integer;
  offs : integer;
  len : integer;
  wstats : integer;
  sbuff : rddfl
end;

state = (CLOSED, OPEN, WFCC, WFTRESP, WFCN); { states of TPRENTA }

orXsts = (open, closed, inprogress);

waessXsts = (zero, one, two, three, four);

waessXerr = (five, six, seven, eight, nine, ten, eleven,
  twelve, thirteen, fourteen, fifteen, sixteen, seventeen, eighteen,
  nineteen, twenty, twentyone);

idss = integer;

efnn = integer;

tadd = integer;

tnnt = integer;

wndnoo = 1..NOPTRI

arr = packed array[1..2] of char
38  procedure werra(pTrwnd : pstdi;
39    rJcase : integer;
40    mes : wmessXerr);
41  var
42    i : integer;
43    j : integer;
44
45  begin
46    writeln(tty, 'entering werra procedure');
47
48    case mes of
49      → (% select the proper error status %)
50      five:
51        begin
52          writeln(tty, 'IMPOSSIBLE SITUATION');
53          writeln(tty, 'IMPOSSIBLE SITUATION')
54          end; (% five %)
55
56      six:
57        begin
58          writeln(tty, 'ILLEGAL USER REQUEST IN 'CLOSED' STATE');
59          writeln(tty, 'ILLEGAL USER PRIMITIVE CODE IS', ord(pTrwnd).fix[1])
60          end; (% six %)
61
62      seven:
63        begin
64          writeln(tty, 'ILLEGAL USER REQUEST IN 'OPEN' STATE');
65          writeln(tty, 'ILLEGAL USER PRIMITIVE CODE IS', ord(pTrwnd).fix[1])
66          end; (% seven %)
67
68      eight:
69        begin
70          writeln(tty, 'ILLEGAL USER REQUEST IN 'WFCC' STATE');
71          writeln(tty, 'ILLEGAL USER PRIMITIVE CODE IS', ord(pTrwnd).fix[1])
72          end; (% eight %)
73
74      nine:
75        begin
76          writeln(tty, 'ILLEGAL USER REQUEST IN 'WFTRESP' STATE');
77          writeln(tty, 'ILLEGAL USER PRIMITIVE CODE IS', ord(pTrwnd).fix[1])
78  end;
end; (* nine *)

ten:
begin
  writeln(tty, 'ILLEGAL USER REQUEST IN 'WFNC' STATE');
  writeln(tty, 'ILLEGAL USER PRIMITIVE CODE IS', ord(#trum1, fixp[1]));
end; (* ten *)

eleven:
begin
  writeln(tty, 'ER TPDU RECEIVED FROM TPRENTB');
  writeln(tty, 'CAUSE FOR REJECTION OF TPDU IS ', reason);
  writeln(tty, 'THE ERRONEOUS PACKET RETURNED BY TPRENTB IS');
  for ji = 0 to 7 do (* assuming that ERRLEN = 128 *)
  begin
    for i := ((ji * 8) + 1) to ((ji + 1) * 16) do
      writeln(tty, ord(#trum1, ath[10#i]));
  end;
  writeln(tty, 'NETWORK DISCONNECTION REQUEST SENT TO NTRNSDA');
end; (* eleven *)

twelve:
begin
  writeln(tty, 'INVALID 'CC' TPDU RECEIVED FROM TPRENTB');
  writeln(tty, 'CAUSE FOR REJECTION OF TPDU IS ', reason);
  writeln(tty, 'NETWORK DISCONNECTION REQUEST SENT TO NTRNSDA');
end; (* twelve *)

thirteen:
begin
  writeln(tty, 'ER TPDU RECEIVED FROM TPRENTB');
  writeln(tty, 'CAUSE FOR REJECTION OF TPDU IS ', reason);
  writeln(tty, 'THE ERRONEOUS PACKET RETURNED BY TPRENTB IS');
  for ji = 0 to 7 do (* assuming that ERRLEN = 128 *)
  begin
    for i := ((ji * 8) + 1) to ((ji + 1) * 16) do
      writeln(tty, ord(#trum1, ath[10#i]));
  end;
  writeln(tty, 'NETWORK DISCONNECTION REQUEST SENT TO NTRNSDA');
end; (* thirteen *)
fourteen:
  begin
  writeln(tty, 'ILLEGAL TPDU RECEIVED IN "CLOSED" STATE FROM TPRENTB');
  writeln(tty, 'THE CODE OF THE TPDU IS ', ord('trwn9', fixp[1]));
  writeln(tty, 'TPDU IS IGNORED')
  end; (* fourteen *)

fifteen:
  begin
  writeln(tty, 'ILLEGAL TPDU RECEIVED IN "WFNC" STATE FROM TPRENTB');
  writeln(tty, 'THE CODE OF THE TPDU IS ', ord('trwn9', fixp[1]));
  writeln(tty, 'TPDU IS IGNORED')
  end; (* fifteen *)

sixteen:
  begin
  writeln(tty, 'ILLEGAL TPDU RECEIVED FROM TPRENTB IN "WFTRESP" STATE');
  writeln(tty, 'THE CODE OF THE TPDU IS ', ord('trwn9', fixp[1]));
  writeln(tty, 'TPDU IS IGNORED')
  end; (* sixteen *)

seventeen:
  begin
  writeln(tty, 'ERRORNEOUS DT TPDU RECEIVED FROM TPRENTB IN "OPEN" STATE');
  writeln(tty, 'THE CODE OF THE TPDU IS ', ord('trwn9', fixp[1]));
  writeln(tty, 'TPDU IS SENT TO TPRENTB')
  end; (* seventeen *)

eighteen:
  begin
  writeln(tty, 'ILLEGAL TPDU RECEIVED FROM TPRENTB IN "WFCC" STATE');
  writeln(tty, 'THE CODE OF THE TPDU IS ', ord('trwn9', fixp[1]));
  writeln(tty, 'TPDU IS SENT TO TPRENTB')
  end; (* eighteen *)
nineteen:
  begin
  writeln(tty, 'INVALID "CR" TPDU RECEIVED FROM TPRENTB IN "CLOSED" STATE');
  writeln(tty, 'CAUSE FOR REJECTION OF TPDU IS ', rclose);
  writeln(tty, 'DR" TPDU SENT TO TPRENTB')
  end; (* nineteen *)

end; (* case *)
PASCAL PDF-11 VERSION 6.07  1908-08-04  04:51:57  PAGE 9

WERRA.PAS

168  writeln(tty, 'leaving werra procedure')
169  end.

NO ERROR DETECTED
TOTAL PROGRAM SIZE  007504
OUTERHOST DATA SIZE  000002
RESERVED STACK & HEAP  000530
COMPILATION TIME 47 SECONDS

PAS WERRA:WERRA/P42-WERRA
NAME: WMESS

FUNCTION: To write a NTRNSDA error message on the terminal in case illegal requests are made by TPRENFA or NTRNSDB.

USAGE: Procedure WMESs(messno: integer;
      stateXa: state;
      rrea: netXrea) external

INPUT PARAMETERS: messno - the type of message;
stateXa - the state of NTRNSDA,
rrea - the inputed string.

OUTPUT PARAMETERS: NONE

MODULES CALLED: (The symbol '>' designates a primitive)
>halt
>writeln

EXIT CONDITIONS: Normal exit conditions.

REMARKS: NONE

*****************************************************************************

(*$INTRNSDA.DEF)
const
FIXPLEN = 8;
OTHLEN = 130;
NETREQLEN = 61;
AFRNO = 71;
WDSZ = 13;
REGSZ = 21; { NOPTR times WDSZ }
NOPTR = 71;
DUHLEN = 10;
DUMMLEN = 5;
NODATFLO = 21;
TERMAFLAG = 42;
TERMBFLAG = 40;
TRNSDAFLAG = 43;
TRNSDBFLAG = 41;
CRCODE = 224;
CCCODE = 208;
DRCODE = 128;
DTCODE = 240;
ERCODE = 112;

type
std = record
  fixp : packed array [1..FIXPLEN] of char;
  othp : packed array [1..OTHLEN] of char;
end;

psstd = "std";

cont1 = array[0..7] of integer; { format for CRGS Executive Directive }
netXrea = packed array[1..NETREQLEN] of char;

ddd = record
  reu : netXrea;
  dum : array[1..DUHLEN] of integer;
end;

add = "ddd";
tskk = record
  nam : array [1..2] of integer;
procedure wmess(messno: integer; stateXa: state; irreu: netXreu);

const
  video = 'C0m';
  up    = 'CA';

begin
  (* Put in reverse video display mode *)
  writeln(chr(27), video, chr(27), up);

  if irreu = 'ustat' then
    begin
      write(tty, 'NTRNSDA is in ');
      case stateXa of
        one: write(tty, 'IDLE ');
        two: write(tty, 'OUT CONNECT PENDING ');
        three: write(tty, 'IN CONNECT PENDING ');
        four: write(tty, 'DATA TRANSFER READY ');
        end;
      writeln(tty, 'state.');
    end
  else case messno of
    1: writeln(TTY, 'eccon');
    2: writeln(TTY, 'trind');
    3: writeln(TTY, 'tdind');
    4: writeln(TTY, 'lnodi');
    5: writeln(TTY, 'lexdi');
    6: begin
      write(tty, 'NTRNSDA cannot accept ', irreu, ' in ');
      case stateXa of
        one: write(tty, 'IDLE ');
        two: write(tty, 'OUT CONNECT PENDING ');
        three: write(tty, 'IN CONNECT PENDING ');
        four: write(tty, 'DATA TRANSFER READY ');
        end;
      writeln(tty, 'state.');
    end
  end;
75        end;    (* case *)
76    end;    (* put back in normal mode *)
77    writeln(chr(27), 'v', chr(27), 'u');
78  end.
79
80  end.

NO ERROR DETECTED
TOTAL PROGRAM SIZE   002336
OUTERHOST DATA SIZE  000002
RESERVED STACK & HEAP 000520
COMPILATION TIME 22 SECONDS

PAS WHESS,WHESS/P42=WHESS
NAME: WOAB

FUNCTION: To send information to task NTRNSDB via system directives
send data (SDAT$) or send by reference (SREF$).

USAGE: procedure waab(ptrwnd : psld;
   var wndno : wndno;
   nxrea : netxrea;
   id : integer;
   bol : boolean); extern;

   waab(ptrwnd, wndno, nxrea, id, bol);

INPUT PARAMETERS:
   ptrwnd - the pointer to the buffer to be sent.
   wndno - the window number to be used if bol = false.
   nxrea - the string to be sent.
   id - the region id.
   bol - the boolean switch.

OUTPUT PARAMETERS:
   wndno - the next window number.

MODULES CALLED:
   (The symbol "->" designates a primitive)

   ->halt
   ->writeln
   ->craw
   ->sref
   ->sdat

EXIT CONDITIONS:
If there are any errors detected in calling CRAW$, SDAT$ or SREF$, then an error message identifying the error and
error code is printed onto standard I/O and the processor
is then halted by using the 'halt' primitive.

The return code of all these system directives is set or
greater or equal to 0 (zero) in case of successful completion
and less than 0 in case of unsuccessful completion.
For more information on the use of these system routines and their return codes, see the RSX-11M V3.2 Executive Reference Manual.

REMARKS: NONE

***

(*1T+FRENTA.DEF)
```
const
FIXPLEN = 8;
OTHLEN = 130;
STRLEN = 118;
NETREDLEN = 61;
CRCODE = 224;
CCCODE = 208;
DTCODE = 240;
DRCODE = 128;
NRDRCODE = 120;
ERCODE = 112;
APRNO = 71;
WNDSZ = 31;
REGSZ = 211; { NOPTR times WNDSZ }
NOPTR = 71;
DUNLEN = 10;
DUMMLEN = 5;
NODATFLG = 20;
IVCODE = 193;
DVCODE = 224;
ERRLEN = 128;
GCDEST = 2222;
GCSOUR = 1111;
TPDUSZPL = 1;
TPDUSZFC = 192;
THPC = 137;
GCTPDU = 128;
GCTHR1 = 11;
GCTHR2 = 21;
GCTHR3 = 31;
GCTHR4 = 41;
THPL = 12;
TRDPL = 81;
TRDLC = 136;
GCTRDL = 71;
GCTRDL2 = 81;
GCTRDL3 = 91;
GCTRDL4 = 101;
NREOT = 99;
GCREOS = 111;
UNDEF = 99;
```
44  type
45    std = record
46      ( GLOBAL TYPES )
47      ( TPDU$ )
48      fixp : packed array [1..FIXPLEN] of char;
49      oth : packed array [1..OTHLEN] of char;
50    end;
51    rstd = "std";
52    contrl = array[0..7] of integer; ( format for CRRG$ )
53    netxreq = packed array[1..NETREDLEN] of char;
54    ddd = record
55      rew : netxreq;
56      dummy : array[1..DUMMYLEN] of integer;
57    end;
58    rddd = "ddd";
59    tskk = record
60      nam : array [1..2] of integer;
61    end;
62    rbuff = record ( format for RCVB$ )
63      nam : tskk;
64      rew : netxreq;
65      dummy : array [1..DUMMYLEN] of integer;
66    end;
67    buff = record ( format for SDAT$ )
68      rew : netxreq;
69      dummy : array [1..DUMMYLEN] of integer;
70    end;
71    rddd = record
72      ok : array[1..2] of integer;
73      rew : netxreq;
74      dummy : array[1..DUMMYLEN] of integer;
75    end;
76    rddd = "rddd";
```pascal
87    rwdbb = record ( format for CRAW$ and RREF$ )
88       rarr : packed array[1..2] of char;
89       rvbadd : pstd;
90       rsiz : integer;
91       rresid : integer;
92       roffs : integer;
93       rlen : integer;
94       rwstats : integer;
95       rsrbuf : rddi;
96    end;
97
98    wdbb = record ( format for CRAW$ and SREF$ )
99       arr : packed array[1..2] of char;
100      vbadd : pstd;
101      siz : integer;
102      resid : integer;
103      uoffs : integer;
104      len : integer;
105      wstats : integer;
106      srbuf : rddi;
107    end;
108
109    state = (CLOSED, OPEN, WFCC, WFTRESP, WFNC); // states of TPRENTA
110
111    opXsts = (opens, closed, inprogress);
112
113    wmessXsts = (zer, one, two, three, four);
114
115    wmessXerr = (five, six, seven, eight, nine, ten, eleven,
116            twelve, thirteen, fourteen, fifteen, sixteen, seventeen, eighteen,
117            nineteen, twenty, twentyone);
118
119    idss = integer;
120
121    efnn = integer;
122
123    tmsg = integer;
124
125    tntt = integer;
126
127    wndnns = 1..NOPTRI;
128
129    arr = packed array[1..2] of char;
```
54
55  procedure buffclear(efn: efnn); extern;
56  procedure buffdo(efn: efnn); extern;
57  procedure buffwait(efn: efnn); extern;
58
59  procedure wdb(var ptrwnd: pstd;
60       var wndno: wndno;
61       nXrea: nXrea;
62       id: integer;
63       bol: boolean);
64
65  var
66      bf : buff;
67      efn : efnn;
68      ids : idss;
69      isrb : ddd;
70      swdb : wdbbi;
71      tsk : tsKK;
72      wdb : wdbbi;
73
74  procedure crw(var wdb : wdbbi; var ids : idss); extern(fortran);
75  procedure send(var tsk : tsKK;
76      var bf : buff;
77      var efn : efnn;
78      var ids : idss); extern(fortran);
79
80  procedure sref(var tsk: tsKK;
81      var efn : efnn;
82      var swdb : wdbbi;
83      var isrb : ddd;
84      var ids : idss); extern(fortran);
85
86  begin
87    writeln(tty, ' entering wdb procedure');
88
89  (* set up task name = 'TRNSDB' *)
90    tsk.name[1] := 32734;
91    tsk.name[2] := 30662;
92
93  if bol then (* send data *)
begin
  (* put string in buffer to be sent *)
  bf.rea := nXrea;
begin
  (* send the data to the TRNSDB task *)
  send(tsk, bf, elm, ids);
if ids < 0 then
  begin
    writeln(tty, 'error in calling SEND in SENDDB, error code = ', ids);
    halt;
  end;
else
  begin (* send by reference to task TRNSDB *)
    (* create a send window in the region *)
    wdb.apr[2] := chr(APRND); (* Active page register 7 *)
    wdb.siz := WNDSZ;
    (* the size of the window in 32-word blocks *)
    wdb.reid := id;
    (* region id *)
    (* or 0 for task region *)
    wdb.offs := (wndno-1)*WNDSz;
    (* specified only if WS.HAP is set *)
    wdb.len := WNDsz;
    (* length to map (32W blocks) *)
    (* or 0 if the length is to default to either
the size of the window or the space remaining
in the region *)
    wdb.wstats := 386h;
    (* window status word *)
    (* bits set: *)
    WS.64B = 256
    WS.HAP = 128
    WS.WRT = 2
    cra(wdb, ids);
  end;
  writeln(tty, 'window created ids = ', ids);
  writeln(tty, 'id assigned to the window = ', ord(wdb.apr[1]));
  writeln(tty, 'virtual base address = ', ord(wdb.vbadd));
  writeln(tty, 'length actually mapped in 32-word blocks = ', wdb.len);
  writeln(tty, 'window status word = ', wdb.wstats);
if ids < 0 then
  begin
    writeln(tty, 'error in using cra, error code = ', ids);
  end;
halt
end;

(* stick buffer into the send window *)

wnd.vbadd".fix" := ptrwnd".fix";
wnd.vbadd".oth" := ptrwnd".oth";

(* prepare window definition block for send by reference *)

swdb.wid := id;     (* id of the region to be sent by reference *)
swdb.offs := (wndno-1) * WNDSZ;  (* offset in region (32W blocks) *)
swdb.len := WNDSZ;   (* length to map (32W blocks) *)
swdb.wstats := 3;    (* window status word *)
                     (* bits set: *)
                     (* WS.WRT = 2  *)
                     (* WS.RED = 1  *)

(* put string into the control send by reference buffer *)

isrb.rea := nXrea;

sref(tsk, sfn, swdb, isrb, ids);

{ writeln(tty, 'ids for sref is ', ids); }

if ids < 0 then
begin
  writeln(tty, 'error in using sref, error code = ', ids);
  halt
end;

(* fix up window number to point to the next window in the circular queue *)

if wndno = NOPTR then
  wndno := 1
else
  wndno := wndno + 1;

end;

{ writeln(tty, 'leaving wgab procedure'); }

end.

NO ERROR DETECTED.
TOTAL PROGRAM SIZE    001744
OUTERMOST DATA SIZE   000002
RESERVED STACK & HEAP 000642
COMPILATION TIME 36 SECONDS
**NAME:** WTERMA

**FUNCTION:** Write an appropriate legal message on the terminal A which could be TCIND, TDIND, THODI or TCCON with appropriate parameter values for the above primitives, as per ISO Transport Protocol Spec. Apr. 1983.

**USAGE:** Procedure wtera(strnd: pstdi;
mes: wmessxls); external

wtera(strnd, mes);

**INPUT PARAMETERS:**
strnd - the pointer to the TPDU received,
mes - the message number.

**OUTPUT PARAMETERS:** NONE

**MODULES CALLED:** (The symbol '\*\*' designates a primitive)
>char
>ord
>writeln
val

**EXIT CONDITIONS:** Normal exit conditions.

**REMARKS:** A lot of the code is for generating a nice display on a VT100 compatible terminal.

Most of the fields in a TPDU are outtrated using the 'ord' primitive. This is because the fields are stored in byte (character) format.

The VAL function is used to associate two bytes together and form an integer.
```
const
  FIXPLEN = 8;
  OTHLEN = 130;
  STRLEN = 118;
  NETREQLEN = 6;
  CRCODE = 224;
  CCCODE = 208;
  DTCODE = 240;
  DRCODE = 128;
  NDRCODE = 120;
  ERCODE = 112;
  APRNG = 7;
  WNDYSZ = 3;
  RESYSZ = 21;  \{ NOPTR lined WNDYSZ \}
  NOPTR = 7;
  DUHLEN = 10;
  DUMMLLEN = 5;
  NODATFFLG = 20;
  IVCODE = 193;
  DVCODE = 224;
  ERKLEN = 128;
  GCDEST = 2222;
  GCSOUR = 1111;
  TPDUSZPL = 1;
  TPDUSZPC = 192;
  TPHP = 137;
  GCTPDU = 128;
  GCTHR1 = 1;
  GCTHR2 = 2;
  GCTHR3 = 3;
  GCTHR4 = 4;
  THPL = 12;
  TRDPL = 8;
  TRDPC = 136;
  GCTRD1 = 7;
  GCTRD2 = 8;
  GCTRD3 = 9;
  GCTRD4 = 10;
  NREOT = 99;
  GCRESO = 11;
  UNDEF = 99;
```
type

   { GLOBAL TYPES }

   std = record
      TPDU: packed array[1..TPDUL] of char
   end;

   { std = "std" }

   contl = array[0..7] of integer; { format for CRG$ }

   netXreq = packed array[1..NETXREQLEN] of char;

   ddd = record
      reu: netXreq;
      dum: array[1..DUMLEN] of integer;
   end;

   { ddd = "ddd" }

   tskk = record
      nam: array[1..2] of integer;
   end;

   rbuff = record { format for RCVD$ }
      nam: tskk;
      reu: netXreq;
      dum: array[1..DUMLEN] of integer;
   end;

   { / buff = "buff" }

   buff = record { format for SDAT$ }
      reu: netXreq;
      dum: array[1..DUMLEN] of integer;
   end;

   rddd = record
      uk: array[1..2] of integer;
      reu: netXreq;
      dum: array[1..DUMLEN] of integer;
   end;

   { rddd = "rddd" }

   rddd = rddd;
rwdbb = record (format for CRAH# and RREF#)
  rarr : packed array[1..2] of char;
  rvbadd : std;
  rsiz : integer;
  rresid : integer;
  roffs : integer;
  rlen : integer;
  rstats : integer;
  rsrbuf : std;
end;

wdbb = record (format for CRAH# and SREF#)
  warr : packed array[1..2] of char;
  vbadd : std;
  six : integer;
  wresid : integer;
  woffs : integer;
  wlen : integer;
  wstats : integer;
  wsrbuf : std;
end;

state = (CLOSED, OPEN, WFCC, WFTRESP, WFNC) (states of TPRENTA)

opXsts = (opens, closes, inprogress);

wmessXsts = (zeron, onetwothree,four);

wmessXerr = (fivesixseven,eigth,nine,ten,eleven,
  twelvethirteen,fourteen,fifteen,sixteen,seventeen,eighteen,
  nineteen,twentynINETWENTYONE);

idss = integer;
efnn = integer;
tadd = integer;
ttt = integer;
wndno = 1..NOPTRI;
arr = packed array[1..2] of char;

91 writeln(tty, 'CALLED TSAP-ID :: ', ord(ptrwand.oth[5]));
92 writeln(tty, 'TPDU SIZE :: ', ord(ptrwand.oth[10]));
93 writeln(tty, 'THROUGHPUT :: ',
94 val(ptrwand.oth[11], ptrwand.oth[12]),
95 val(ptrwand.oth[13], ptrwand.oth[14]),
96 val(ptrwand.oth[15], ptrwand.oth[16]),
97 val(ptrwand.oth[17], ptrwand.oth[18]));
98 writeln(tty, 'TRANSIT DELAY :: ',
99 val(ptrwand.oth[27], ptrwand.oth[28]),
100 val(ptrwand.oth[29], ptrwand.oth[30]),
101 val(ptrwand.oth[31], ptrwand.oth[32]),
102 val(ptrwand.oth[33], ptrwand.oth[34]));
103 writeln(tty);
104 writeln(tty, 'do you accept this TRANSPORT CONNECTION ??');
105 writeln(tty, 'if you do please type in a TCRES with appropriate parameters');
106 writeln(tty, 'or type in a TREQ with appropriate parameters if you ');
107 writeln(tty, 'wish to disconnect');
108 end;
109 (% one %)
110 two:
111 begin
112 writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty);
113 writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty);
114 writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty);
115 writeln(tty, '< TRANSPORT CONNECT CONFIRMATION >>');
116 writeln(tty, 'RESPONDING ADDRESS :: ',
117 val(ptrwand.fixp[5], ptrwand.fixp[6]));
118 (% destination address not required as per primitive parameter definition %)
120 writeln(tty);
121 writeln(tty, 'quality of service parameters are :: ');
122 writeln(tty, 'CALLING TSAP-ID :: ', ord(ptrwand.oth[4]));
123 writeln(tty, 'CALLED TSAP-ID :: ', ord(ptrwand.oth[5]));
124 writeln(tty, 'TPDU SIZE :: ', ord(ptrwand.oth[10]));
125 writeln(tty, 'THROUGHPUT :: ',
126 val(ptrwand.oth[11], ptrwand.oth[12]),
127 val(ptrwand.oth[13], ptrwand.oth[14]),
128 val(ptrwand.oth[15], ptrwand.oth[16]),
129 val(ptrwand.oth[17], ptrwand.oth[18]));
130 writeln(tty, 'TRANSIT DELAY :: ',
131 val(ptrwand.oth[27], ptrwand.oth[28]));
134 val(ptrwd,o[29]; ptrwd,o[30]);
135 val(ptrwd,o[31]; ptrwd,o[32]);
136 val(ptrwd,o[33]; ptrwd,o[34]));
137 writeln(tty);
138 writeln(tty, 'TRANSPORT CONNECTION is now OPEN');
139 writeln(tty, 'Please type in user data using the TNODR primitive with');
140 writeln(tty, 'appropriate parameters OR type in a TDREQ with appropriate');
141 writeln(tty, 'parameters if you wish to disconnect');
142 end;
(* two *)
143 three:
144 begin
145 writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty);
146 writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty);
147 writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty);
148 writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty);
149 writeln(tty, '<< TRANSPORT DATA INDICATION >>');
150 writeln(tty);
151 writeln(tty, 'the data sent in by the responder is :');
152 writeln(tty);
153 writeln(tty, chr(27), chr(5), chr(27), chr(65), chr(27), chr(69));
154 write(tty,
155 ptrwd,o[3], ord(ptrwd,o[1]) + 2));
156 writeln(tty, chr(27), chr(5), chr(27), chr(65), chr(27), chr(69));
157 writeln(tty);
158 writeln(tty, 'Please type in user data using the TNODR primitive with');
159 writeln(tty, 'appropriate parameters');
160 writeln(tty, 'OR type in a TDREQ with appropriate parameters if you');
161 writeln(tty, 'wish to disconnect');
162 end;
(* three *)
163 four:
164 begin
165 writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty);
166 writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty);
167 writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty);
168 writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty);
169 writeln(tty, '<< TRANSPORT DISCONNECT INDICATION >>');
170 writeln(tty);
171 writeln(tty, 'the TRANSPORT CONNECTION is now disconnected :)');
172 writeln(tty);
173 writeln(tty, 'the reason for disconnection is :');
174 writeln(tty, ord(ptrwd,fix[8]));
175 writeln(tty, 'additional information for disconnection is :');
177    writeln(tty);
178    writeln(tty, chr(27), '5m', chr(27), 'A');
179    write(tty,
180        ptrwnd.oth[3,(ord(ptrwnd.oth[1]) + 2)]);
181    writeln(tty, chr(27), '0m', chr(27), '7m', chr(27), 'A');
182    writeln(tty); writeln(tty); writeln(tty); writeln(tty); writeln(tty)
183        (** four **) { (**) case **) }
184    end;
185 (** put back in normal video **)
186    writeln(tty, chr(27), '0m', chr(27), '724m', chr(27), '71f', chr(27), 'A');
187    writeln(tty, 'leaving wterma procedure');
188    end;

NO ERROR DETECTED
TOTAL PROGRAM SIZE 015026
OUTERMOST DATA SIZE 000002
RESERVED STACK & HEAP 000520
COMPILATION TIME 131 SECONDS

PAS WTERMA/WTERMA/P42-WTERMA