1

1.0

1.1

1.25

1.5
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

1.4

2.8

3.2

3.6

4.0

2.5

2.2

2.0

1.8

1.6

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Services des thèses canadiennes

Ottawa, Canada
K1A 0N4

CANADIAN THESES

THÈSES CANADIENNES

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.

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 microfilmés

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

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

National Library
of Canada

Bibliothèque nationale
du Canada

CANADIAN THESES
ON MICROFICHE

0-315-22179-8
THÈSES CANADIENNES
SUR MICROFICHE

NAME OF AUTHOR/NOM DE L'AUTEUR ALEXANDER S. PEPPE

TITLE OF THESIS/TITRE DE LA THÈSE LOGICAL DESIGN AND TESTING

OF OSI SOFTWARE

FOR UNIX

UNIVERSITY/UNIVERSITÉ CARLTON UNIVERSITY

DEGREE FOR WHICH THESIS WAS PRESENTED/
GRADE POUR LEQUEL CETTE THÈSE FUT PRÉSENTÉE M. ENG.

YEAR THIS DEGREE CONFERRED/ANNÉE D'OBTENTION DE CE GRADE 1984

NAME OF SUPERVISOR/NOM DU DIRECTEUR DE THÈSE PROF. R. J. A. DUMR

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

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

L'autorisation est, par la présente, accordée à la BIBLIOTHÈ-
QUE NATIONALE DU CANADA de microfilmer cette thèse et
de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la
thèse ni de longs extraits de celle-ci ne doivent être imprimés
ou autrement reproduits sans l'autorisation écrite de l'auteur.

DATED/DATE Jan 31, 1984 SIGNED/SIGNÉ Alex Pepple

PERMANENT ADDRESS/RÉSIDENCE FIXE 25 HEGAN ST, #8

NEPEAN, ONT K2E-5E8

LOGICAL DESIGN AND TESTING OF
OSI SOFTWARE FOR UNIX

by

© Alexander Siminiowuna Pepple

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Engineering

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
Ottawa, Ontario
December 1983

The undersigned recommend to the Faculty of Graduate
Studies and Research, the acceptance of the thesis,
"LOGICAL DESIGN AND TESTING OF
OSI SOFTWARE FOR UNIX"
submitted by Alexander Siminiowuna Pepple, B.Eng.,
in partial fulfillment of the requirements for the degree of
Master of Engineering.

.....*E. J. B. B.*.....
Thesis Supervisor

.....*B. P. R.*.....
Chairman,
Department of Systems and
Computer Engineering

Carleton University
December, 1983

ABSTRACT

This thesis describes the design, implementation and testing of the transport and session layers of a gateway system based on CCITT's (International Telegraph and Telephone Consultative Committee) Teletex protocols, under the UNIX operating system.

A detailed data flow inspired analysis of Teletex is presented including an investigation of its ISO extensibility potentials. Also investigated are suitable UNIX-based protocol system architectures with portability and configurability characteristics. Following this, the details of the adopted system structure and of its implementation are given. A key feature of the adopted design approach is that it is based on interaction exchange - a messaging scheme between the major system modules. This design approach is compared to other approaches.

Protocol system testing is another major topic covered. A test script-oriented protocol testing scheme based on interaction passing, and protocol extension of the layer under test is described followed by the design and implementation of the test system. Also given are the testing of the Teletex implementation using this test system, and a comparison of the test approach to other approaches.

ACKNOWLEDGEMENTS

The author wishes to express his thanks to the following people for their contributions towards this thesis:

- his thesis advisor, Dr. Ray Buhr, whose continued guidance and inspired suggestions were invaluable in developing and consolidating the ideas that went into this thesis;

- the members of the OSI gateway project team, Chris Griffiths and William Malek - they are a great team to work with;

- Fred Kaudel for his excellent proofreading;

- and last but not least, his wife, Mary for her infinite patience, understanding and constant encouragement.

TABLE OF CONTENTS

Abstract	
Acknowledgements	
Table of Contents	
List of Figures	
List of Tables	

CHAPTER 1 INTRODUCTION

1.1	Genesis	1-1
1.2	Goals	1-2
1.3	Accomplishments	1-2
1.4	Related Work	1-3
1.5	Outline	1-4

CHAPTER 2 - SYSTEM SPECIFICATIONS AND REQUIREMENTS

2.1	OSI Overview	2-1
2.2	Teletex	2-3
2.2.1	Overview	2-3
2.2.2	Basic Teletex	2-5
2.2.2.1	Teletex Session/Document Levels	2-5
2.2.2.2	Teletex Transport Level	2-6
2.2.3	Extended Teletex	2-7
2.3	Teletex versus ISO's Session and Transport Layers	2-7
2.4	Nature of the Implemented Session and Transport Layers	2-10
2.5	Design Goals	2-11
2.5.1	Non-Functional Requirements	2-11
2.5.2	Customizability	2-12
2.6	External System Interfaces	2-12

CHAPTER 3 LOGICAL DESIGN OF THE SYSTEM

3.1	Methodology	3-1
3.2	Protocol and Service Data Flows	3-1
3.3	Major System Components and Data Flows	3-16

CHAPTER 4 DETAILED DESIGN AND IMPLEMENTATION

4.1	Introduction	4-1
4.2	UNIX and Communication Systems	4-1

4.3	High Level System Characteristics	4-5
4.3.1	Local Flow Control	4-5
4.3.2	Logical System Structure	4-5
4.3.3	Application Interface	4-6
4.4	The Teletex Process	4-10
4.4.1	Teletex Pseudo-processes	4-12
4.4.1.1	Structure	4-12
4.4.1.2	Some Implementation Details	4-15

CHAPTER 5 SYSTEM TESTING

5.1	Introduction	5-1
5.2	System Under Test	5-1
5.3	Test Strategy	5-2
5.3.1	Overview	5-2
5.3.2	Test Interface	5-4
5.3.3	Test Service Data Units	5-4
5.3.4	Test Protocol Data Units	5-5
5.3.5	Test Evaluation	5-5
5.4	Test System Configuration and Architecture	5-8
5.4.1	Configuration	5-8
5.4.2	Architecture	5-13
5.5	Implementation Details	5-13
5.5.1	Test System Initiation and Operation	5-13
5.5.2	Test Sequences	5-15
5.6	Discussion	5-16
5.6.1	Evaluation of the Test Scheme	5-16
5.6.2	Other Approaches	5-17
5.7	Conclusion	5-18

CHAPTER 6 DISCUSSION

6.1	Meeting Design Goals	6-1
6.1.1	System Structure	6-1
6.1.2	Testing Approach	6-2
6.2	Other Approaches	6-2
6.3	Multiple Sessions	6-6
6.4	Other Issues	6-6
6.4.1	UNIX performance	6-6
6.4.2	Teletex FSMs	6-8

CHAPTER 7 CONCLUSION

7.1	Design Reflections	7-1
7.2	Status of Project	7-3
7.3	Further Work	7-3

BIBLIOGRAPHY

GLOSSARY OF TERMS

APPENDIX A GRAPHICAL NOTATIONS

APPENDIX B TELETEx PDU ABBREVIATIONS

APPENDIX C TELETEx FSMs

**APPENDIX D DOCUMENTATION FOR TRANSPORT LAYER SOFTWARE
TESTING**

APPENDIX E SAMPLE OF THE TEST CODE

LIST OF FIGURES

2.1	Hierarchical Layout of the ISO Reference Architecture Layers	2-2
2.2	Layered Protocol Architecture Data Units Relationships	2-4
2.3(a)	Functional Comparison of ISO and Teletex Transport Layers	2-8
2.3(b)	Functional Comparison of ISO and Teletex Session Layers	2-9
2.4	The Environment of the Teletex Gateway System	2-13
3.1	Major Design Tactics	3-2
3.2	Format for Data Flow Illustrating Protocol/Service Logic: Connection Establishment Example	3-4
3.3(a)	Data Flow Sequence for Successful Connection Attempt	3-9
3.3(b)	Sample Logic for Layer n Connecting Actions	3-10
3.3(c)	Transport Layer Initiation of Connection Refusal	3-11
3.3(d)	A/P Layer Initiation of Connection Refusal	3-12
3.4	Data Flow Sequence for Data Transfer and Synchronization	3-13
3.5	Data Flow Sequence Showing Handling of Provider Error	3-14
3.6	Major Components and Format of Interaction Flow Between them	3-17
4.1	Interprocess Messaging Strategies Using UNIX pipes	4-3
4.2	System Structure	4-7
4.3	Internals of the Teletex Process	4-8
4.4	Structure and Mechanisms of the Application Interface Module	4-9
4.5	Pseudo Code for MAIN	4-11
4.6	(n)-Pseudo Process' Internals	4-13
4.7	n_PT Pseudo Code Logic	4-14
4.8	Form of States Definition	4-17
5.1	Test Strategy Explained by Data Flows and Pseudo Code	5-3
5.2	Test System Configurations	5-10
5.3	Test System Structures	5-12
5.4	Internals of the TM_Process	5-14

6.1	Procedure-Oriented Structure of the Teletex System	6-4
6.2	Possible Configuring of the Teletex System for Multiple Sessions	6-7

LIST OF TABLES

3.1	Session Service Primitives	3-5
3.2	Transport Service Primitives	3-6
3.3	Teletex Network Service Primitives	3-7
5.1	Test SDUs	5-6
5.2	Test PDUs	5-6
5.3	Test Primitives Description	5-7
5.4	Test Evaluation Scheme: A Test Score Sheet	5-9

CHAPTER 1

INTRODUCTION

1.1 GENESIS

The International Standards Organization (ISO) in an attempt to standardize the interconnection scheme for dissimilar computers and networks, introduced the Open Systems Interconnection (OSI) standard which is characterized by seven protocol layers [6]. These OSI standards are still evolving and some of their aspects have not been finalized. Various protocols and service definitions already exist for many of the layers especially the lower ones, and they are mainly from ISO and CCITT (Consultative Committee on International Telegraph and Telephone) [6-15]. The protocols concern themselves only with the interworking between peer protocol entities while the service definitions deal with the interfacing of layers within the same computer system. Ultimately, the implementor is given a free hand in the design details of an actual implementation. Since OSI standards are fairly new, there is currently an apparent dearth of example implementations of OSI protocol systems, and hence, an inadequate supply of practical guides on implementation and testing approaches and means for resolving the associated design issues.

Consequently, there is clearly a need for researching and developing a methodical approach to the design, implementation and testing of OSI-inspired protocol systems. That there is a need for a design methodology for OSI systems is buttressed by the fact that complex communications protocol systems designed and implemented using ad hoc techniques are more often than not error-prone and may exhibit unexpected and undesirable behaviour.

Currently, intensive research activities are under way at Carleton University in various aspects of the OSI standards. One of these activities is concerned with the design and implementation of a UNIX-based OSI gateway system using the TransCanada Telephone System (TCTS) version of CCITT's Teletex protocols (session and lower levels) [16]. This thesis is an offshoot of this project, its primary concern being the analysis and description of the system's design, implementation and testing under UNIX.

1.2 GOALS

The immediate goal of the project is to provide a gateway from the Carleton local area network (Ethernet) to Datapac (X.25). The Carleton gateway environment [2] consists of: 1) a VAX 11/750 computer with 485 megabytes of hard disk; 2) four SUN microsystem workstations with Motorola 68000 processors; 3) Berkeley 4.2 UNIX supporting the VAX, and 4.1c supporting the SUNs [3,4]. The OSI gateway is located in one of the SUN machines and consists of portable session and transport layers written in C [5] and running under the standard UNIX subset of Berkeley 4.2 (4.2 will be available on the SUNs by the spring of 1984). It is planned to provide low level communication support with an X.25 network board whose software will be written in PLM/86 and run under RMX/86.

The objectives of this thesis are:

To investigate and propose a design approach for portable, configurable and extensible OSI gateway systems;

To investigate ways of suitably using the system development tools of the UNIX operating system for protocol systems design, implementation and testing;

To design and implement the Teletex-based OSI gateway system (session and transport levels) for the UNIX operating system from the principles so obtained;

To evolve a practical test scheme for protocol systems and use it in testing the implementation obtained.

1.3 ACCOMPLISHMENTS

The project was a cooperative effort involving a group of students. So far the high level design of the system has been completed and documented. The transport layer software has been fully coded and tested in a stand-alone mode, and more than half of the session layer has been coded and compiled. The author's contributions to the project were:

- A clear description of the Teletex protocols in a form suitable for implementation.

- A detailed Teletex/ISO functional comparison leading to an ISO-extensible implementation of Teletex.

- A detailed data flow view of the protocol leading to the definition of inter-layer interfaces and the mapping out of implementation-related non-functional system requirements.

- Developing portions of the architecture of the Teletex gateway process and of its transport and session layer pseudo-processes and their implementation details; and partially writing the session layer software.

- Evolving a suitable test scheme for the Teletex system. This included an investigation of protocol extension in order to maximize the controllability and status monitoring of the layers to be tested, leading to the definition of test service data units, protocol data units and a corresponding test FSM. The evolved test scheme is script-driven and based on interaction exchanges with the layer under test.

- The design of the test system and its implementation. The author also used this implementation to fully test in a stand-alone mode the transport layer software written by another project participant.

1.4 RELATED WORK

There is currently an apparent shortage of work directly dealing with UNIX-based OSI systems. However, much can be found on OSI systems in general. There is existing work on determining and analyzing design issues for OSI systems. Some typical examples can be found in [17-19].

There is currently an ongoing design of a Teletex system at the University of British Columbia under G. Neufeld. Though there is no published material on this activity at the present, it has been gathered through private communication that the system design is "procedure-oriented". Also recently, Buhr and MacKinnon [21] have presented a scheme for evolving towards OSI from pre-OSI facilities, using an inter-library file transfer system as an example. Other related work in the area of OSI system design and implementation can be found in [19,20,28].

There is also some existing work on OSI system testing. Bochmann and Cerny [25,26] have presented a method for extending protocol definitions and service specifications to facilitate testing, and have also described example test structures and an actual implementation. Other examples of work on testing can be found in [23,24].

1.5 OUTLINE

Chapter 2 gives a summary of system design requirements. This includes an overview of the Teletex protocol and service definitions, ISO's session and transport level protocols and services, and how to make Teletex extensible with respect to ISO's standards. This chapter also gives the external view of the system, specifically, the operator interface.

Chapter 3 describes the high level design of the system. It starts with a description of the design methodology including the perspective, strategy, tactics and mechanisms. It then proceeds to give data flow and other high level descriptions of the system.

Chapter 4 which gives the detailed design and implementation of the system starts with an analysis of the UNIX operating system with a view to mapping out candidate system architectures that satisfy the data flow patterns of the previous chapter. The final architecture is then chosen and following this are internal details of the system including the implementation issues and problems.

Chapter 5 describes the test scheme for the system. Included is a description of the test strategy, the test system configuration and architecture, and design details of the test system. The test approach is also evaluated and compared to other approaches.

In chapter 6, a general discussion and evaluation of the design and test approaches and how these meet initial objectives are given. Included is a look at other approaches to the design of UNIX-based OSI systems.

The concluding chapter highlights key issues of the thesis, provides a status report on the project and details further work to be done.

CHAPTER 2

SYSTEM SPECIFICATIONS AND REQUIREMENTS

2.1 OSI OVERVIEW

The OSI reference architecture defines seven protocol layers (fig. 2.1) working independently and cooperatively to achieve interconnection of computer systems [6]. A given layer provides certain services to the next higher layer by building on the services obtained from the layer immediately below to which it adds more facilities. This is not totally applicable to the extreme layers, namely, the application layer that serves the end-user and the physical layer that uses the facilities of the underlying transmission medium. More extensive discussions on protocols can be found in [27,28].

Two interconnected n -layer entities (peer entities) interact logically using an n -level protocol. They intercommunicate through the exchange of n -level protocol data units (PDU). Two broad PDU categories can be identified: a) control PDUs used for protocol control and management dealing with functions like connection establishment and release, flow and error control, and synchronization; b) information PDUs used for transferring user data.

Physically, the n -layer peer entities use the services of the $(n-1)$ -layer for the actual data transfer between them. Thus the n -layer is the user of the $(n-1)$ -layer service, or alternatively, the $(n-1)$ -layer is the service provider to the n -layer. The local interface between the n - and the $(n-1)$ -layer is specified by the $n-1$ service primitives composed of the $n-1$ interface control information which they use for coordinating their activities, and the $n-1$ interface data which is really data exchanged by the n peer entities through the services of the $(n-1)$ -layer. The interface control information is used for connection establishment or release,

LEVEL

Fig. 2.1 - HIERARCHICAL LAYOUT OF THE OSI REFERENCE ARCHITECTURE LAYERS

synchronization and error control. Typically, the (n-1)-layer adds a header containing (n-1)-level protocol management information to the n-level user data before sending it to the other side. This header is stripped off by the n-1 entity at the other side before it is sent up. The amount of the n-1 interface data preserved from one n-1 level entity to the other is a service data unit (SDU). Figure 2.2 illustrates the inter- and intra-system interworking just described.

Teletex specifies the session and lower layers. This thesis deals with the session and transport layers only.

2.2 TELETEX

This section presents a brief look at the Teletex services and protocols. Starting with a Teletex overview, it proceeds to give a description of Basic and Extended Teletex.

2.2.1 Overview

Teletex is a business communications service used by dissimilar office text machines for communicating with one another. Teletex is a set of recommendations from CCITT [10-14]. TCTS has provided service details for Teletex in [16]. This TCTS version which is compatible with CCITT's recommendations is what is used for this project. Teletex was chosen because it was well defined at the start of the project and because of its extensibility with respect to ISO protocol standards: its transport is analogous to ISO transport class 0 and ISO's session layer includes Teletex' S.62 functionality (as a result of ISO's research).

Teletex consists of recommendations F.200, S.60, S.61, S.62, and S.70. The first three recommendations are mainly concerned with hardware requirements for Teletex terminals and application/presentation related functions. This includes things like quality of service, networking requirements, text handling provisions, storage requirements, graphical control character repertoires, coding, etc. The recommendations of interest for this project are S.62 - describing the control procedures for the Teletex session/document layers, and S.70 - describing the control procedures for Teletex's network

Fig. 2.2 - LAYERED PROTOCOL ARCHITECTURE DATA UNITS RELATIONSHIPS

independent transport layer [13,14,16]. Some elements of these control procedures are currently incomplete.

2.2.2 Basic Teletex

2.2.2.1 Teletex Session/Document Levels

These are defined by recommendation S.62. Overall, these levels are responsible for establishing, managing and terminating a dialogue (session) between two application/presentation entities. The purpose of this dialogue is for the exchange of Teletex documents. In a Teletex context, a document is a sequence of one or more pages while a page is roughly equivalent to a standard office page of text.

The terminal that initiates session connection is the initiator (calling side) while the other is the respondent (called side). In Basic Teletex the dialogue mode between two terminals may be One Way Connection (OWC) or Two Way Alternate (TWA). There is a data token which must be acquired before any of the two entities engaged in the dialogue can send data to the other. The token possessor (source) may send data to the other side and the latter (sink) may only send acknowledgements. In the TWA mode, the call initiator has the data token at the onset, but the source/sink relationship between the two sides can change dynamically in an unrestricted fashion during the lifetime of the session. OWC is a subset of TWA and the source/sink relationship is fixed throughout the session, with the caller always in possession of the data token.

The Teletex session/document specifications describe two groups of related procedures - the "session" procedures and the document control procedures. Analysis reveals that the document procedures are in fact a subset of the session procedures: the session functions are for connection establishment and release, activity management and error recovery; the document level functions deal with the details of activity management. One session connection maintains only one transport connection at any given time.

Some details of the functions performed by the Teletex session level are the following:

Connection Establishment: This includes identification of the relevant CCITT service (Teletex, Facsimile [16] etc.); terminal identification; date and time; session identification; non-basic session and terminal capabilities negotiation.

Session Management: This includes management of the data token, synchronization and error reporting. Errors may be due to local terminal error, failure of the transport connection, procedural error, or some undefined reason. All of these will cause an abnormal termination of the session. There are no error recovery mechanisms.

Session Release: This function is used for clearing the session connection and optional clearing of the transport connection.

Document level procedures deal with activity management functions. An activity is a logical piece of work performed during a session. In Teletex an activity is primarily concerned with the transmission of a complete document from the source to the sink. Some of the functions associated with activity management are: activity initiation/termination and activity synchronization/resynchronization. Activity initiation includes document referencing and possibly, some terminal capabilities negotiation. Activity synchronization primarily involves checkpointing. A checkpoint (activity mark in ISO terminology) is a reference number inserted within the text stream to assist in error recovery. Resynchronization functions are related to error recovery.

2.2.2.2 Teletex Transport level

This is analogous to ISO's Basic Transport Class 0 [9]. Some functions performed by the Teletex transport layer are: connection establishment that includes transport connection identification; data delimitation; detection and reporting of procedural error or network connection failure. No error recovery mechanisms are provided.

There is no end-to-end flow control. This is performed at the network/link levels as appropriate. Furthermore, no

explicit transport connection release function is available, hence the lifespan of the transport connection is dependent on that of the network connection.

2.2.3 Extended Teletex

Extended Teletex defines some non-basic functions for quality of service enhancement. Some example of such enhancements are: the Two Way Simultaneous (TWS) session mode where each of the two interconnected terminals can be source and sink simultaneously; and, a session suspension facility which can be used for clearing the call for a negotiable time without losing the relationship represented by the session. Other enhancements may also be achieved by the negotiation of non-basic parameters, e.g. being able to use more than just default values, typically, a session window size other than 3, or, a transport block size other than 128 bytes. The latter may necessitate transport level segmentation/concateration.

2.3 TELETEX VERSUS ISO'S SESSION AND TRANSPORT LAYERS

The S.62 session/document levels just described is non-standard in ISO's context since on the one hand, ISO does not explicitly identify any document level functions, and on the other hand, the S.62 session and document levels procedures do not exhibit a clean functional separation. However, recent collaboration on the part of ISO and CCITT, centered on the integration of S.62 and ECMA-75 have resulted in the definition of a generalized ISO basic session (BS) level protocol and service which accomodates S.62 functionality and yet is S.62 independent [7,8]. Some functions available in this BS includes error recovery and an expedited data service among others. Besides, it incorporates all S.62 session and document functions. Thus, Teletex is a subset of this BS and application rules are given on how to derive the Teletex state machine from it; and, the names of the S.62 commands/responses have been changed to conform to ISO terminology, though their encodings have been retained. figures 2.3(a) and (b) show that Teletex' functionality is hierarchically related to ISO's session and transport layers. ISO's names have been used for the functions and they are described in [7,8].

Fig. 2.3(a) - Functional Comparison of ISO and Teletex Transport Layers

Fig. 2. (b) - Functional Comparison of ISO and Teletex Session Layers

In view of ISO's accommodation of S.62 in its BS, and the fact that the Teletex transport is analogous to ISO's Basic Transport Class 0, there is every indication that making Teletex extensible with respect to ISO's protocol standards should be straightforward. This fact is buttressed by the hierarchies represented in figures 2.3(a) and (b).

The approach to extensibility adopted here mainly entails use of ISO's terminology instead of S.62 names whenever possible. Also, Basic Teletex is implemented since the specification of most of the functions of Extended Teletex is still incomplete. Only default values for the parameters are used leaving room for the addition of enhanced features of Extended Teletex, or indeed, of ISO's BS, since there is no major restrictions to a Basic S.62 implementation being upgraded in a step-wise fashion to a full ISO session layer.

The general design philosophy has been to parameterize features in Basic Teletex which may be subject to negotiation, e.g. window size, data block size etc. even if the current system only uses the default parameter values. Being able to add extra system/protocol capabilities with minimal system modifications requires a logical structure that readily accommodates the addition of new features and functions.

2.4 NATURE OF IMPLEMENTED SESSION AND TRANSPORT LAYERS

The project implements TWA Basic Teletex and provides only default parameter values. Dealing only with default parameter values does not affect the ability of the system to interwork with other Teletex systems since an ability to use only default values is required of all implementations, and the use of other parameter values is optional and subject to negotiation. Some of these features present in the current implementation are:

Session: A fixed, non-negotiable window size of 3 Teletex pages, and session level data blocks of less than 128 bytes. The latter enables a maximum transport PDU size of 128 bytes and does away with any need for segmentation/combination at the transport level.

It should be noted that the session layer is not responsible for buffering/retransmission. This is the responsibility of the session service (SS) user. The SS passes minor and major synchronization point numbers and acknowledgements to the user to assist the latter in carrying out this function. Essentially, the SS provides session connection/disconnection, activity management, and error reporting services.

Transport: Like the SS, the transport service (TS) uses only default parameter values. No buffering is needed for this level since data requests to the TS are in transport-sized blocks of less than 128 bytes. Also no retransmissions are done at this level (this function is handled at the network/link level).

The basic services provided by this layer to the session level are transport connection establishment which includes transport connection identification; transport connection release (using the services of the network layer); data transfer, and error detection and reporting.

2.5 DESIGN GOALS

2.5.1 Non-functional Requirements

The design goals for the UNIX-based Teletex protocol system are the following:

- Use of only standard UNIX features is required to enable portability with respect to all versions of UNIX;

- Also required is a logical system structure that is portable with respect to other operating systems like RSX/11, CP/M, RMX/86 etc. Besides, translating the C software code into other languages like ADA, PLM/86, PASCAL etc. should be straightforward. Details on these portability requirements can be found in [33];

- System configurability is desired. This mainly entails a logical system structure flexible enough to be readily adaptable to different hardware/software/protocol environments;

- Extensibility of the Teletex implementation with respect to ISO protocol and service standards is desired. The key aspects of this requirement have already been discussed in section 2.3.

2.5.2 Customizability

Figure 2.4 shows the software environment for the Teletex gateway system. It interacts with application/presentation (A/P) processes above, and with an X.25 network board below it. A network board driver is required for interfacing to the X.25 board, and there is also a need for an operating system dependent interface to the A/P processes. Thus, customization of these interfaces for different operating system environments is required. Here, only the UNIX environment is of interest.

Conceptually, the A/P processes make service requests to the Teletex system using the SS primitives (given in table 3.1). The nature of the A/P process or of their protocols is not specified by Teletex and is outside the scope of this thesis. The only details given here is how these processes can initiate the Teletex system and interact with it.

Currently, the X.25 board is under a separate development and not yet completed, and details of its driver interface are not available at the moment. However, on a higher conceptual level, the Teletex system can be thought of as interfacing to the network service (NS) of the X.25 board via the network service primitives (table 3.3).

2.6 EXTERNAL SYSTEM INTERFACES

This concerns the external interfaces provided to the A/P processes (users) by the Teletex system in a UNIX operating system environment. The purpose of these interfaces are to free the users of details of system initiation/shutdown and data transfer mechanisms (fig 2.4). In fact, these interface specifications are operating system independent and not restricted to UNIX since they are merely procedure calls, resulting in structural portability of user interface. Details

Fig. 2.4 - THE ENVIRONMENT OF THE TELETEx GATeWAY SYSTEM

of these procedures will however be operating system dependent.

To be able to use the Teletex system the user software has to be compiled with the interface procedures file - INIT_TELETEX. A positive integer value is returned from the procedures if all went well, otherwise a negative value is returned. Here is a description of the procedures:

Startup: The startup procedure creates the Teletex processes and all the required UNIX pipes (for interprocess messaging), configuring them properly for i/o. Its C definition is:

```
int START_TTX();
```

Operation: These operation procedures are used to send/receive interactions to/from the Teletex system. They hide the piping details from the user. Note that since standard UNIX pipe i/o calls are eventually made by the procedures, they are blocking in nature. Thus on input calls, the caller will be blocked until there is something to receive. This is not true for output since there is a local 'credit' mechanism which should be used to ensure that interactions are sent to the Teletex system only when the required credit is available. All pointers passed to these procedures should refer to buffers with a minimum size of 128 bytes since the Teletex system uses only fixed-size interactions of this value. The C definitions for the i/o procedures are:

```
int GET_INTERACTION( in_buff_ptr)
    char *in_buff_ptr;
```

```
int PUT_INTERACTION( out_buff_ptr)
    char *out_buff_ptr;
```

Shutdown: A call to the shutdown procedure results in termination of the Teletex processes. Its C definition is:

```
int STOP_TTX();
```


CHAPTER 3

LOGICAL DESIGN OF THE SYSTEM

3.1 METHODOLOGY

The logical design perspective for a protocol system, expressed or implied, is that of a hierarchical set of layered modular entities working cooperatively and independently for the provision of some well-defined services to an end user [6,17,21]. Achieving the aims of a design perspective requires the adoption of a matching set of design strategy tactics and mechanisms [32].

An "edges-in" design strategy was adopted for the design of the Teletex system since this strategy is consistent in yielding the desired results for the design of various types of system, communication systems included [30-32]. In this strategy, the overall external view of the system, characterized by its external interfaces (at the "edges") is first mapped out, then by successive and hierarchical increase in details inwards, the full system structure is laid out.

Design tactics outline the system development steps. Figure 3.1 gives the major tactics for the development of the Teletex system. The design of the test system is recursively in line with steps 2 to 5 of the figure.

Design mechanisms, are the tools used for the design activity and system description. Structure and data flow graphs (described in appendix A), and some pseudo code as necessary are used here.

3.2 PROTOCOL AND SERVICE DATA FLOWS

The first step of the tactics of fig. 3.1 are involved here. So the data flows between peer entities (in terms of

Fig. 3.1 - Major Design Tactics

PDUs) and between the system layers (in terms of SDUs) for primarily the session and transport layers are mapped out. Whenever necessary, the relationship between SDUs and the corresponding PDUs are established. The purpose of this analysis is to obtain a better comprehension of the protocols under implementation, map out peculiar data flows, and establish protocol-dependent non-functional requirements.

Service primitives may be confirmed or non-confirmed. The confirmed ones may be of the request, indication, response or confirmation types, while the non-confirmed ones may be requests or indications only. Some responses or confirmations can only be positive while others can also be negative. Figure 3.2 shows the form of these data flows for two systems, using connection establishment primitives as examples.

Teletex protocol analysis involved mapping out the following data flow elements:

- 1) S.62 session/document service primitives and associated PDUs;
- 2) S.70 TS primitives and associated PDUs;
- 3) X.25 NS primitives.

Items 1) to 3) are detailed in tables 3.1 to 3.3. respectively. Included in these tables are the uses of the primitives and, for extensibility purposes, corresponding ISO names are provided.

The protocol analysis technique used here utilizes special data flow structures to map out on a gross scale the sequence of inter-layer data exchanges within each system, and the consequent peer-to-peer data exchanges from one system to the other. Some typical examples of such data flow structures are given here (Teletex names and abbreviations have been used for shorter labelling; S.62 and S.70 PDU abbreviations used are explained in appendix B):

NOTE: RESPONSE/CONFIRMATION may be positive or negative.

Fig. 3.2 - FORMAT FOR DATA FLOW ILLUSTRATING PROTOCOL/SERVICE LOGIC:
CONNECTION ESTABLISHMENT EXAMPLE.

TABLE 3.1 - Session Service Primitives

<u>SERVICE</u>	<u>ISO NAME</u>	<u>S.62 NAME</u>
1. Session Connection Establishment	S-CONNECT	S-CONNECT (a)
2. Orderly Session Release	S-RELEASE	S-DISCONNECT (b)
3. User-Initiated Session Abort	S-U-ABORT	S-ABORT (b)
4. Normal Data Exchange	S-DATA	S-DATA (e)
5. Session Failure Reporting	S-P-EXCEPTION REPORT	S-FAILURE (d)
6. Session Error Reporting	S-P-EXCEPTION REPORT	S-ERROR (d)
7. Token Management	S-TOKEN GIVE	S-CHANGE CONTROL (b)
8. Session Synchronization	S-SYNC MINOR	S-SYNC MINOR (a)
9. Capability Data	S-CAPABILITY DATA	S-SELECT(b)
10. Activity Management	S-ACTIVITY BEGIN (start)	S-START (c)
	S-ACTIVITY BEGIN (continue)	S-CONTINUE (c)
	S-ACTIVITY END	S-SYNC MAJ (a)
	S-ACTIVITY DISCARD	S-DISCARD (b)

S.62 notes:

- (a) Response/Confirmation may be positive or negative.
- (b) Only positive Response/Confirmation defined.
- (c) No positive Response/Confirmation sent but negative Response/Confirmation sent if necessary.
- (d) Provider-initiated.
- (e) Non-confirmed.

TABLE 3.2 - Transport Service Primitives

<u>SERVICE</u>	<u>ISO NAMES</u>	<u>S.70 NAMES</u>
1. Transport Connection Establishment	T-CONNECT	T-CONNECT (a)
2. Orderly Transport Release	T-DISCONNECT (request)	T-DISCONNECT (b)
3. Normal Data Exchange	T-DATA	T-DATA (b)
4. Transport Error Reporting	T-DISCONNECT* (indication)	T-ERROR (c)

S.70 Notes: (a) Confirmed positive or negative.
(b) Non-Confirmed
(c) Provider-Initiated.

- * - ISO has no T-ERROR SDU;
- DISCONNECT reason will be given as T-ERROR.

TABLE 3.3 - Teletex Network Service Primitives

<u>SERVICE</u>	<u>TELETEX NAMES</u>	<u>TYPES</u>
1. Network Connection Establishment	N-CONNECT	Confirmed (positive or negative)
2. Network Connection Orderly Release	N-DISCONNECT	Non-confirmed
3. Network Data Transfer	N-DATA	Non-confirmed
4. Transport Error Reporting	N-ERROR	Provider- Initiated

- Figure 3.3(a) gives the data flow sequence for a successful connection establishment attempt. Notice the sizeable amount of intra and inter system handshaking needed to initiate a session connection. Figure 3.3(b) is a pseudo code showing the common logic for connection establishment derived from figure 3.3(a). Figures 3.3(c) and 3.3(d) give the data flow patterns for connection refusal initiated at the transport and the application layers respectively. The pattern for refusal initiation at the session level can be similarly obtained. The flows for data transfer and synchronization protocol functions are mapped out in figure 3.4

- Figure 3.5 shows the data flow sequences for dealing with provider errors. Data flows for resynchronization, data token management and connection release can be similarly obtained and have been omitted here for brevity.

These data flows and protocol analysis lead to the following decisions about the non-functional requirements for the Teletex system:

Degree of Autonomy: There should be some capability for concurrent events handling, autonomous processing, and spontaneous initiation of certain actions from the Teletex layers in view of the following facts:

- Full-duplex operation is required of the Teletex system since S.62 control procedures require sending a document while at the same time being able to receive acknowledgements for the transmitted data, and vice-versa. This calls for a capability, real or virtual, for concurrent handling of data transmission and reception. Real concurrent processing requires separate transmit and receive processes; for the virtual case, a single process does it all. As it will be shown, this design uses just a single Teletex process, this being adequate enough for the degree of sophistication of Teletex.

- Some spontaneous actions may be taken by the session or transport layers. This is because provider-initiated errors (e.g. connection failure) and timeouts (e.g. inactivity and command session abort timeouts) requires a provider to spontaneously send provider-initiated error-reporting primitives to the service user.

NOTE: All RESPONSES/CONFIRMS are positive

Fig. 3.3(a) - DATA FLOW SEQUENCE FOR SUCCESSFUL CONNECTION ATTEMPT

CALLING SIDE

(n-ESTABLISH REQUEST just
obtained from n layer):

- 1) Send (n-1)-ESTAB. REQ.
to (n-1) layer;
- 2) Wait for (n-1)-ESTAB. CONF.
from layer (n-1);
- 3) Send n-CONNECT PDU to peer;
- 4) Wait for n-ACCEPT PDU
from peer;
- 5) Send (n)-ESTAB. CONF. to
layer (n+1).

CALLED SIDE

((n-1)-ESTABLISH INDICATION
just received from
(n-1) layer):

- Send (n-1)-ESTAB. RESP. to
(n-1) layer;
- Wait for n-CONNECT PDU from
peer;
- Send n-ESTAB. IND. to (n+1)
layer;
- Wait for n-ESTAB. RESP.
from (n+1) layer
- Send n-ACCEPT PDU to peer.

Fig. 3.3(b) Sample Logic for Layer n Connecting Actions

Fig. 3.3(c) - TRANSPORT LAYER INITIATION OF CONNECTION REFUSAL

Fig. 3.3(d) - A/P LAYER INITIATION OF CONNECTION REFUSAL

* Several TDT PDU may result from T-level segmentation/combination

Fig. 3.4 - DATA FLOW SEQUENCE FOR DATA TRANSFER AND SYNCHRONIZATION

Fig. 3.5 - DATA FLOW SEQUENCE SHOWING HANDLING OF PROVIDER ERROR

Buffering/Retransmission: No buffering or retransmissions are needed by the Teletex protocol layers. This is because, as can be seen from the inter-layer data flows, no PDUs are sent out by either the transport or session layers unless specifically requested by the service user, be it for connection, data transfer, synchronization or resynchronization. For instance in fig. 3.5, when the session layer is notified of transport connection failure through the T-ERROR INDICATION primitive, it sends an S-FAILURE INDICATION to the SS user and does nothing else until the user requests an abort using the S-ABORT REQUEST primitive. Only then is the CSA PDU sent out.

Thus, the session and transport layers basically process each user requests completely, immediately sending out any resulting outputs.

No segmentation/concatenation is done at the transport or session layer. Thus they do not need to store outgoing data for splitting nor partially assembled incoming data. Furthermore, retransmissions are not in the domain of the Teletex layers. The SS sends synchronization point numbers and acknowledgements to the SS user to assist the latter with any buffering/retransmission functions it may have. Note however that extensions to more sophisticated ISO protocols may require the addition of this functionality.

Local Flow Control: That there is no buffering infers that no local flow control is required. The protocol elements of procedure provide adequate mechanisms for end-to-end flow control to the SS users through checkpointing: The sink does not need to acknowledge a checkpoint unless there is enough space for about another receive-window of pages. The source only receives check point acknowledgements from the sink, so it has no critical buffering, and consequently, flow control needs.

However, a redundant local flow control mechanism was added to the design. This is mainly for extensibility purposes as this facilitates the upgrading of the system to handle the more sophisticated ISO protocols which may require local flow control. The flow control mechanism used is in the form of inter-layer two-way credit interactions. Details on this are eventually given.

3.3 MAJOR SYSTEM COMPONENTS AND DATA FLOWS

The major components of the Teletex system are shown in figure 3.6. This figure also identifies the major inter-module data flows.

A key feature of this structure is the fact that the interworking between the major system modules is by way of interaction-passing (inter-module messaging) via the input software ports of the destination modules. Thus interactions can be thought of as a set of logical interfaces between the system modules.

The modules concerned may be active (e.g. processes) or passive (e.g. Ada-style passive packages). A module sending an interaction to some destination module is in effect providing the latter with a work notice, requiring the latter to perform some specific processing or action.

Two major classes of interactions can be identified: service interactions - related to the service definition between layers; and local interactions - dealing with local processing functions. Service interactions embody all the service primitives (tables 3.1 to 3.3) while local interactions involve things like time (for timeout purposes), and local flow control. Being credit-based, the flow control scheme fits naturally with the interaction-passing mechanism.

Some advantages to using interactions are flexibility and extension potentials. All interface requirements/functions can be readily encoded as interactions. Adding new interface elements or functions will merely involve, in general, new entries to tables of interactions definition, and of course adding to the system the logic for dealing with the new functions. Also, interaction passing can be used in any operating system environment. Specifically, it naturally blends with the UNIX environment where interprocess communication is message-oriented.

The flow pattern of figure 3.6 is one in which each system layer has an input port for receiving interactions from the neighbouring layers. These are processed and depending on

Fig. 3.6 - Major Components and Format of Interaction Flow Between Them.

the results, output interactions may be sent up or down or both ways to the input ports of the adjacent layers. For configurability, the input/output port mapping between the major system entities should be easy to change (possibly through recompiling or dynamically). This may be necessary for accomodating different hardware/software system configurations, like moving some of the layers of Teletex (e.g. the transport layer) to hardware.

In the next chapter, the nature of the system layers and of their input ports for UNIX will be detailed.

CHAPTER 4

DETAILED DESIGN AND IMPLEMENTATION

4.1 INTRODUCTION

In this chapter, the details of the Teletex system architecture are given. This involves steps 3 and 4 of the tactics of figure 3.1.

First, features of the UNIX process environment are reviewed to determine how these could suitably be used for communication systems development. From this analysis, candidate architectures for the Teletex system are explored leading to the adoption of the most suitable one.

Eventually, details of the logical structure of the adopted system architecture are given in conjunction with the pertinent implementation details.

4.2 UNIX AND COMMUNICATION SYSTEMS

In a UNIX context, a process is the execution of an image. An image is a computer execution environment and includes the program, the associated data, the open files and the current directory [1-4].

The only way a new UNIX process can be created is by an already existing one, the parent, forking to create a child process, and this can be repeated hierarchically in the child. A child inherits its parent's process image, in other words, it is an exact copy of the parent (except for having a different process identification). Thus the new process will execute the same piece of code as its parent unless it makes an exec call to a different program. An exec is analogous to a procedure call from which there is no return.

UNIX processes cannot share the same data structure. Each has its own private data area. Thus, data transfer by reference instead of by value between processes, which is a popular means of making communication software more efficient is useless in UNIX; and neither can monitors [30] be used for interprocess communication since this also requires memory sharing.

Pipes and signals are the only mechanisms for interprocess communication in UNIX. signals are basically software interrupts, while pipes are a form of interprocess messaging channels or software ports maintained by the operating system. For our purpose, only pipes are of interest. In standard UNIX, one limitation of pipe i/o is the fact that it is blocking in nature, with no timeouts provisions. An attempt to read from an empty pipe results in the calling process being blocked until some other process writes to the pipe. Similarly, a process trying to write to a full pipe will be blocked until space is created by another process reading from it.

The blocking nature of pipe i/o has some important repercussions for UNIX-based communication system design. In a communication environment, a process may need to wait for multiple events (e.g. incoming and outgoing messages) simultaneously. It would be unwise to make such a process wait on multiple events at multiple pipes, especially if the events are infrequent in nature, because if it gets blocked on one input pipe then it cannot attend to other pipes which may contain events needing servicing. This may result in long service delays and consequent performance degradation.

This problem can be dealt with in two ways as illustrated in figures 4.1(a) and 4.1(b).

In figure 4.1(a), a special event pipe is provided. The process only waits at this pipe for notification of input availability at its message pipes which it will then proceed to pick up. After servicing this event, it will return to the event-notice pipe to wait for more notifications. Other processes sending to this process first write the message to the appropriate pipe and then write to the event pipe, thus

Fig. 4.1 - Inter-process Messaging Strategies Using UNIX Pipes

notifying the waiting process of the message, including the identity of the pipe involved.

In the second approach (fig. 4.1(b)), the process has only one input pipe where it waits for interprocess messages. Messages will have a field identifying the source processes.

The first approach has the disadvantage of being cumbersome and requiring relatively more memory since it uses more pipes. Also, it takes more time to pick up an interaction since two pipe accesses instead of one are involved.

On the other hand, the second approach is simpler conceptually and implementation-wise and requires less memory and pipe access time. This approach has been adopted for the present design and it is adequate enough to handle all the interprocess messaging needs of the Teletex system.

Some other unattractive characteristics of UNIX is the fact that pipe i/o is very slow as operating system kernel call overheads are involved. Also, in a multi-user environment there will be multi-user processing delays. This, coupled with the fact that processes may be swapped in and out of main memory into disks, will result in considerable overall system delays. These delays cannot be overlooked for an operation as time-critical as communications. Thus some rules are needed on ways of efficiently designing UNIX-based communication systems.

From the foregoing exploration of the pertinent UNIX tools, the following rules of thumb about the non-functional requirements a UNIX-based communications system should have in order to exhibit an acceptable level of performance were reached:

- 1) A minimum of piping between the protocol layers;
- 2) A minimum of protocol-related processes;

Rule 1 follows from the fact that pipe i/o is slow as kernel calls are involved. Application of this rule involves in part limiting to one the number of input pipes per process, and generally reducing the overall number of system pipes.

Rule 2 results from the need to minimize interprocess data transfer delays: since processes cannot share memory and have to exchange data by value, the more layers are integrated into a single process, the less their data transfer overheads will be since they are now able to share memory, able to pass data between them by reference, and overall make fewer kernel calls.

4.3 HIGH LEVEL SYSTEM CHARACTERISTICS

4.3.1 Local Flow Control

Local flow control takes the form of a two-way credit interaction. Each layer sends interactions up or down only when there is credit to do so. Because of the flow-through nature of the system's data flows due to the absence of buffering, the credit exchange works as follows:

Credit for upwards data flow (UP_credit) -

When the A_process gives UP_credit to the S-layer, this triggers the chain of actions: 1) S_layer gives UP_credit to T_layer; 2) T_layer gives UP_credit to the network.

Credit for downwards data flow (DOWN_credit) -

When the network gives DOWN_CREDIT to the T_layer, this triggers the chain of events: 1) T_layer gives DOWN_credit to S_layer; 2) S_layer gives DOWN_credit to A process.

At startup, the A_process and the network initiate the credit exchanges.

4.3.2 Logical System Structure

From the guidelines for UNIX protocol system design given in section 4.2, coupled with the requirement for structural portability with respect to operating system, some candidate system structures, compatible with an interaction-passing scheme can now be explored:

A possible system structure is one in which each of the protocol layers of figure 3.6 is made into an independent processes and its associated input queue becomes a UNIX pipe. Evaluation shows that this architecture violates the rules for UNIX system design given in section 4.2 since there are too many processes and pipes. Besides, this architecture may be difficult to port to non-multitasking operating systems like CP/M, thus failing to meet one of the major non-functional requirements (in section 2.5.1).

The other system structure for the Teletex system is shown in figure 4.2. Here, the session and transport layers are embodied within one process, the `TELETEX_process` (`TTX_process`) and thus made to share just a single input pipe. To make this structure as general-purpose as possible the transport and the session layers within the `TTX_process` are made to be pseudo-processes (figure 4.3). This means that these layers are designed so that they can be readily converted into full-fledged processes should such a need arise (as a result of porting to an operating system environment with more efficient multitasking facilities like ADA or RMX/86).

The enveloping shell provided by the `TTX_process`' control logic, `MAIN` basically serves as a middle-man for interaction exchanges between these pseudo-processes and the other active system entities (the `A_process` and the network). This structure is relatively more efficient, being in line with the rules for UNIX design provided in the last section: There is less pipe i/o implying less access to the kernel; fewer processes result in decreased data transfer overheads since the `S_` and `T_layer` pseudo-processes can now exchange data by reference as they are within the same UNIX process and can now share memory.

4.3.3 Application Interface

In section 2.6, C procedures were defined for the provision of a Teletex interface to application processes. The structure of this interface is detailed in figure 4.4 including an explanation of the mechanisms these procedures use for implementing their functionality.

Fig. 4.2 - System Structure

Fig. 4.3 - Internals of the Teletex Process

Fig. 4.4 - Structure and Mechanisms of the Application Interface Module.

4.4 THE TELETEX PROCESS

The internal structure of the TTX_process as illustrated in figure 4.3 are as follows:

- An S_layer and a T_layer pseudo-processes each with an input queue S_queue and T_queue respectively.

- Two extra queues, A_queue and N_queue, are reserved for receiving interactions destined for the A_process and the N_board respectively. The S_layer may use the A_queue or T_queue for output depending on whether its output interaction was destined for the A_process or the T_layer. Similarly, the T_layer may send its output interactions to either the S_queue or the N_queue depending on whether they were meant for the S_layer or the network.

- MAIN controls the activities of the TTX_process. These activities include:

- a) Picking up interactions from the TTX_pipe sent by the A_process or the network or the TIMER_process (time interactions) and sending them to the input queue of the destination pseudo-process;

- b) Initiating pseudo-processes processing by calls to their interface procedures, S_Process_Interaction (S_PI) and T_Process_Interaction (T_PI);

- c) Picking up output interactions in the A_queue or the N_queue and sending them to the A_pipe or network respectively.

The pseudo code for MAIN is given in figure 4.5

A strong point of this internal mapping for the TTX_process is its configurability. The pseudo-processes can be readily transformed into real process in a more efficient multitasking environment. The only modifications in the structure needed for this are removing MAIN and the TTX_pipe from the structure and transforming their input queues and the other system pipes to the appropriate interprocess messaging

MAIN

```

begin
  INITIALIZE ALL INTERNAL QUEUES TO 'EMPTY' STATUS;
  do FOREVER
    RECEIVE INTERACTION FROM TTX_PIPE;
    if INTERACTION_TYPE = OUTGOING then begin
      QUEUE INTERACTION IN S_Q;
      S_Q_STATUS := not EMPTY;
    end else begin      /* INTERACTION_TYPE IS INCOMING */
      QUEUE INTERACTION IN T_Q;
      T_Q_STATUS := not EMPTY; end;

    while SOMETHING_TO_PROCESS
      if S_Q_STATUS ≠ EMPTY then
        S_PI; /* MAY ALTER A_Q_STATUS or */
              /* T_Q_STATUS TO not EMPTY */
      if T_Q_STATUS ≠ EMPTY then
        T_PI; /* MAY ALTER S_Q_STATUS or */
              /* N_Q_STATUS TO not EMPTY */

      if N_Q_STATUS ≠ EMPTY then begin
        SEND N_Q_CONTENTS TO NETWORK;
        N_Q_STATUS := EMPTY; end;
      if A_Q_STATUS ≠ EMPTY then begin
        SEND A_Q_CONTENTS TO A_PIPE;
        A_Q_STATUS := EMPTY; end;

    end;
  end;
end;

```

Note: SOMETHING_TO_PROCESS will be true if any of the internal queues of the TTX_PROCESS contain any interactions (determined from the (n+1) and (n-1) Queues status returned from the call.)

Fig. 4.5 - PSEUDO CODE FOR MAIN

channels, like a buffer task in ADA or a mailbox in RMX/86 [33].

4.4.1 Teletex Pseudo-Processes

4.4.1.1 Structure

To ensure that such a reconfiguring is readily carried out in a modular fashion, it is necessary to cleanly separate for each layer protocol-dependent functions from protocol-independent ones.

Thus, as shown in figure 4.6 an n-level pseudo-process contains an interface procedure `n_Process_interaction` (`n_PI`) exclusively responsible for interacting with the pseudo-process' surroundings, and an interactions-processing package `n_Service` that deals with the associated layer's protocol-related functions.

The processing done by `n_PI` include queuing and dequeuing of interactions. These interface procedures have a flexible i/o queue mapping for configurability purposes. Such configuring may be dictated by a different hardware/software system mapping, like moving the transport layer to hardware. Thus these procedures form an enveloping shell which shields the protocol-related package `n_Service` from any knowledge about the operating system environment. The pseudo code logic for `n_PI` is given in figure 4.7.

Continuing the modularization process hierarchically inwards, `n_Service` internals were further demarcated into clear-cut functional components represented by FSMs and the corresponding action procedures. As shown in fig. 4.6, there are protocol/service, testing and the protocol-independent component demarcation. The protocol/service components are devoted to the protocol and service under implementation, while the testing components handle the protocol extensions defined for testing the layer concerned. The protocol-independent components deal with locally supplied functions which are not defined in the protocol under implementation. These functions include the local credit-based flow control and the processing of the time interactions sent in by the `TIMER_process` in order to generate timeout events.

Fig. 4.6 - (n)-Pseudo Process' Internals

n_PI

```
begin
  RECEIVE INTERACTION FROM n_Q;
  n_FUNCTIONS_PROCESSING:
  if OUTPUT INTERACTION AVAILABLE then
    case OUTPUT DIRECTION of
      UP: begin
        SEND TO (n+1)_Q;
        (n+1)_Q_STATUS = not EMPTY;
      end;
      DOWN: begin
        SEND TO (n-1)_Q;
        (n-1)_Q_STATUS = not EMPTY;
      end;
      BOTH: begin
        SEND TO (n-1)_Q;
        SEND TO (n+1)_Q;
        (n-1)_Q_STATUS = not EMPTY;
        (n+1)_Q_STATUS = not EMPTY;
      end;
    end case;
  end;
```

Fig. 4.7 - n_PI PSEUDO CODE LOGIC

Some of the FSMs shown in figure 4.6 comprise a local and an end-to-end component. The end-to-end components deal with peer-to-peer events and actions. These are directly involved with sending and receiving PDUs to and from the other side. The local components deal with the events and actions associated with the adjoining (layers within the same computer system). Thus the protocol-independent FSM and actions have no end-to-end components since they are only concerned with functions within the local system. For the protocol FSMs, the end-to-end states give a high level description of the system's logic while the local states provide the internal details of these end-to-end components.

4.4.1.2 Some Implementation Details

FSM Encoding:

In appendix C, the Teletex FSMs' details have been reproduced from [16] for convenience. In these FSMs, the end-to-end states are the big ellipses while the smaller ellipses contained in these bigger ones are the local states. For instance, in the calling side of the transport FSM, state 0 is an end-to-end state and its local states are 0.1, 0.2 and 0.3. A look at these FSMs shows that there is really no clean separation between local and end-to-end states as some local states are associated with end-to-end events and actions. Nonetheless, a way was found of modularizing protocol FSMs into end-to-end sets with local sub-components. Such a modularization is needed in order to preserve the functional separations represented in these FSMs thus demarcating inter-layer actions and events from peer-to-peer ones.

A table-driven FSM was used used in encoding the FSMs since a table will be easy to modify should it become necessary to enhance the FSMs to deal with ISO protocols. There are at least three ways the FSMs could be encoded:

- 1) Abandon the separation into local and end-to-end components and encode everything as one giant FSM composed of only the local FSM components since it can be seen from Appendix C that all the FSM details (events and actions) whether local or end-to-end always involve local states.

2) This is a variation of approach 1) with some additional ordering introduced: the local states are now grouped into end-to-end sets. Besides, each local state now differentiates between local and end-to-end actions and transitions.

3) The third approach carries the modularization attempted at 2) to the fullest extent possible. Here each state, local or end-to-end is made of a set of outwards and inwards transitions, to and from other states.

Approach 3) was used here since it produces the greatest amount of separation of end-to-end states and events from local ones. Some Pascal-like definitions of the states data structures are given in figure 4.8 with explanatory comments.

Now it is time for some final notes about the FSM encoding and the actions. Error events and actions which are common to a majority of the states are not directly included in their tables of events and actions as this would result in much repetition, leading to a table which is too large relative to the FSM being implemented and hence, rather unmanageable. Hence these common error events and actions are a part of a special error handling FSM state. If the current error event cannot be found in the event list of the current state, an exception handling routine is called and this routine will consult the error handling state for the appropriate actions.

Time Interactions Processing:

No timers are needed by the transport layer protocol machine, but two timers are required by the session/document protocol: the inactivity and the CSA timers. Furthermore, as explained in the next chapter, there is a timer required by the test FSM for testing the timeout functions. The inactivity timer is used to generate a timeout for any period of inactivity greater than 60 seconds. The CSA timer is used to generate a timeout when waiting for a RSAP PDU in response to a previously sent CSA PDU used for aborting the session. The length of the CSA timeout period is not specified in the protocol and hence is assumed implementation dependent.

```
record   LOCAL_STATE =  
    begin  
  
        LOCAL_EVENT           : event_type;  
        LOCAL_ACTION           : action_type;  
        NEXT_LOCAL_STATE       : state_type;  
        GOTO_END_TO_END_STATE : boolean;  
        /* if true, transition to current */  
        /* end-to-end state results */  
  
        LAST_STATE_ELEMENT     : boolean;  
        /* this boolean signals the      */  
        /* last entry in the current state */  
        /* to the table-processing routine */  
  
    end;
```

```
record   END_TO_END_STATE =  
    begin  
  
        LAST_LOCAL_EVENT       : event_type;  
        END_TO_END_EVENT       : event_type;  
        LOCAL_ACTION           : action_type;  
        END_TO_END_ACTION      : action_type;  
        MY_NEXT_LOCAL_STATE     : state_type;  
        NEXT_END_TO_END_STATE  : state_type;  
        LAST_STATE_ELEMENT     : boolean;  
  
    end;
```

Note: Each state whether end-to-end or local is a variable array of records as defined above.

Fig 4.8 Form of States Definition.

To deal with the timing needs, a TIMER data structure (C structure or Pascal record) with TIME_INTERVAL_LENGTH, REMAINING_TIME and STATUS variables is defined. Instances of this data structure are created for each timer needed in the system. To start the timer, STATUS is set to "on" and REMAINING_TIME is set to TIME_INTERVAL_LENGTH. To stop the timer, STATUS is set to "off". REMAINING_TIME is decremented with each time interaction received if STATUS is "on". A timeout event is generated if REMAINING_TIME goes to zero with STATUS still "on".

CHAPTER 5

SYSTEM TESTING

5.1 INTRODUCTION

The test described here is only local confidence testing. This is essentially an advanced debugging stage where the layers of the implementation are locally verified to the point where a good deal of confidence is gained in them and in the system in general. This is in preparation for the next testing stage, involving the use of an active tester (reference implementation) [39,42]. Testing using a reference implementation is not dealt with here because of time limitations associated with the project, but much work in this area can be found in [25,26,34-39].

A key characteristic of the test scheme adopted here is that the layer under test (LUT) is tested as a black box, interfacing to the test system via interactions composed of the standard protocol service primitives and specially contrived test interactions that enable checks on the internal status of the LUT. Protocol extension [25] within the LUT makes such checks possible, besides making the testing more controllable. The test system also interacts with a human test conductor non-interactively via the file system.

The remainder of this chapter begins with a brief description of the system under test and the way it interacts with the testing system. Next, the test strategy is described. This is followed by the test system configuration and architecture and its implementation. Finally, the test approach is analyzed and compared to other approaches.

5.2 SYSTEM UNDER TEST

As described in the last chapter, the system under test consists of UNIX processes interworking via piped interactions. For the test system to fit naturally in this environment, it is made to be just another UNIX process,

henceforth called the Test Manager (TM), with its own input pipe for receiving interactions from the TTX_process (encapsulating the layers).

Under non-testing conditions, the TTX_process interacts with an A_process above it, and the network below. However, under test conditions, the TM takes the place of the application and/or the network depending on the desired test configuration. It is made to present the same interface to the TTX_process and the LUT. Thus, there is no need for either the TTX_process or the LUT to distinguish between a normal or a test environment.

5.3 TEST STRATEGY

5.3.1 Overview

The test strategy is illustrated in figure 5.1. Essentially, the LUT is tested as a black box: the TM applies some excitations to it and observes the consequent responses. In all cases, the excitations and responses are interactions. All features of the LUT are tested according to this scheme, leading to uniformity.

The test system interfaces to the test conductor non-interactively via a previously prepared test script. In the test script, the interactions to be sent to the LUT and the expected responses are specially encoded in the format:

```
[ith SENT INTERACTION] [ith EXPECTED  
INTERACTION] [(i+1)th SENT INTERACTION]  
[(i+1)th EXPECTED INTERACTION]  
.....
```

The TM reads this script from a file and writes the LUT's response interactions to a trace buffer which is dumped to a specified output file when the test session concludes. A ring buffer is used, making it possible to conduct tests with a large output without buffering constraints. Hence, only the most recent trace is available at the end of the test session. The output file is in the format:

CHAPTER. 6

DISCUSSION

6.1 MEETING DESIGN GOALS

From the elaborate descriptions of the UNIX-based system design, implementation and testing presented in the previous chapters, an evaluation can now be carried out on how well the design goals stated at the onset were satisfied. Some of these evaluations have already been presented in the associated chapters so this is mainly a recapitulation.

6.1.1 System Structure

On the question of portability, the evolved system architecture exhibits structural portability. It was necessary to design the UNIX-based Teletex system (session and transport layers) as a single Teletex process composed of a session and a transport layer pseudo-processes for UNIX-related performance reasons. However it was shown that portability benefits accrued from this approach since a single process structure is easy to port to other operating system with limited multitasking facilities like CP/M.

The pseudo-processes had an interface shell dealing with queuing/dequeuing of interactions with a configurable I/O ports mapping, and an interactions processing package for the associated layer's functions whose internals were further modularized into clear-cut protocol-related, testing and protocol-independent components. This modularization ensured that the pseudo-processes could be readily transformed into full-fledged processes if so desired as a result of, for instance, porting to an operating system with better multitasking facilities like ADA, RMX/86 etc. Then, only the interface shell will need modification, and the UNIX-pipes will become RMX/86 mailboxes, ADA buffer tasks and so on.

[ith TEST RESPONSE]
[(i+1)th TEST RESPONSE]
.....

It is an error if the response from the LUT does not match the expected response specified in the script. These erroneous responses are flagged by the TM, making it easy for the test conductor to locate them when the results are analyzed. Generally, a recovery action is taken following such discrepancies, typically, an abort of the test session.

The UNIX standard i/o drivers - open, read, write and close - are sufficient for dealing with all possible i/o needs of the test process.

5.3.2 Test Interface

Besides the regular system interactions, special test interactions (SDUs and PDUs) were added to the system to facilitate monitoring of the LUT's status and activities. To enable the protocol layers implementation deal with these test SDUs and PDUs which are not a part of the standard protocol definition, they were made to incorporate a test FSM. This FSM is separate from the standard FSM that deals with regular interactions. The separation makes for modularity and configurability. Thus, if it be so desired due to efficiency or performance considerations, the test FSM could be readily removed from a tested and certified system. Such reconfiguring could be done at compile time.

When two Teletex systems are coupled in a local-remote configuration, the local TM may send interactions to the remote LUT indirectly using 'remote'-tagged test SDUs. Details on this will eventually be provided.

5.3.3 Test Service Data Units

Here are some examples of the reasoning behind extra SDUs:

- During the initial testing stages (mainly debugging) it may be necessary to check how correctly the LUT adheres to the implemented FSM. This can be done by monitoring its events, its output interactions, its sent and received PDUs, and its state transitions. No extra SDUs are needed by the TM for monitoring the output interactions, but the remaining items are easily monitored by defining a GIVE_STATUS test SDU used by the LUT to report its status; and a SENT_PDU_TRACE and RECEIVED_PDU_TRACE test SDUs used by the LUT to report the PDU last sent and received.

- Consider another scenario. We may want to test a specific function of the local LUT, executed only when a specific PDU is received in a specific state. To shorten the test sequences, we will want to move the local LUT to a specific state without having to go through all the intervening states, and we will also want the remote LUT to send the necessary PDU to its local counterpart. Thus, the FORCE_STATE(state name) and the SEND_PDU(pdu name) test SDUs were defined to meet these needs.

Through a similar reasoning the remaining test SDUs listed in table 5.1 and described in table 5.3 were devised.

5.3.4 Test Protocol Data Units

The local TM cannot send test SDUs directly to the remote LUT, but it can do so indirectly through test PDUs. A remote-tagged test SDU sent to the local LUT by this TM is a way of telling it to send the corresponding test PDU to the remote LUT. Hence, all remote-tagged test SDUs have a corresponding test PDU, while local-tagged ones have none. Table 5.2 gives a list of the test PDUs. The local LUT deals with test SDUs as described in table 5.3, and the remote LUT deals with the related test PDUs in line with the logic for the associated test SDUs given in this table.

5.3.5 Test Evaluation

The analysis of the test result is done at the end of the test session from the resultant output file. An assessment of

 TEST DATA UNITS

Table 5.1 - TEST SDUs:

<u>NAME</u>	<u>TYPE</u>
1. FORCE_STATE.LOCAL(state_name)	Non-confirmed
2. FORCE_STATE.REMOTE(state_name)	Non-confirmed
3. GIVE_STATUS.LOCAL	Confirmed (a)
4. GIVE_STATUS.REMOTE	Confirmed (b)
5. SEND_PDU.LOCAL(pdu_name,param_values)	Non-confirmed
6. SEND_PDU.REMOTE(pdu_name,param_values)	Non-confirmed
7. SENT_PDU_TRACE.LOCAL	Confirmed (a)
8. RECEIVED_PDU_TRACE.LOCAL	Confirmed (a)
9. TEST_TIMEOUT.LOCAL(time_interval)	Non-confirmed
10. TEST_TIMEOUT.REMOTE(time_interval)	Non-confirmed

Notes: (a) Has REQUEST and CONFIRM types only.

(b) REQUEST, INDICATION, RESPONSE, and CONFIRM types.

Table 5.2 - Test PDUs:

1. FORCE_STATE(state_name)
2. GIVE_STATUS.REQUEST
3. GIVE_STATUS.RESPONSE
4. SEND_PDU(pdu_name,parameter_values)
5. TEST_TIMEOUT(time_interval)

Table 5.3 - Test Primitives Description

1. FORCE_STATE.LOCAL(x):
Puts the local LUT into the protocol state x.
Possible use of this would be to shorten the time required to get to a given FSM state of the protocol by not going through the intervening states.
 2. FORCE_STATE.REMOTE(x):
Same as 1. but applies to remote LUT.
 3. GIVE_STATUS.LOCAL:
Causes the local LUT to give a status report to its TM. The status will be primarily the current state of the LUT; other status types may be included as needed.
 4. GIVE_STATUS.REMOTE:
Same as 3. but applies to remote LUT.
 5. SEND_PDU.LOCAL(x,y):
This instructs the local LUT to construct and send PDU x, using the set of values, y for the PDU parameters. y may be omitted, in which case the standard parameters values will be used by the LUT. Also, some or all the values for the parameters may be specified through y.
 6. SEND_PDU.REMOTE(x,y):
Same as 5. but applicable to remote LUT.
 7. SENT_PDU_TRACE.LOCAL:
Causes the LUT to make available to its TM, a copy of the last PDU sent out.
 8. RECEIVED_PDU_TRACE.LOCAL:
This causes the LUT to make available to its TM, a copy of the last PDU received.
 9. TEST_TIMEOUT.LOCAL(x):
Causes the local LUT to go into a state where it ignores all the incoming PDUs from the remote LUT (except for PDU x) thus forcing the latter to generate a timeout. The local LUT leaves the timeout testing state when it receives PDU x, and the necessary action for x is taken as dictated by the protocol FSM.
 10. TEST_TIMEOUT.REMOTE(x):
Same as 9. but applies to remote LUT.
-

the degree of the test's success is obtained from the amount of deviation exhibited by the LUT from expected behaviour. The criteria for determining whether the LUT passed or failed the test can be gathered from the test objectives for an n-level protocol software.

The primary objectives in testing an n-level protocol system implementation are:

a) Verify that all the basic functions specified in the service definition for interfacing to the n+1 layer are correctly provided.

b) Certify that proper interaction with a peer protocol entity through the correct exchange of PDUs is maintained.

c) Verify the proper use of the services of the n-1 layer provider by the correct exchange of service primitives, and adequate reactions to provider-initiated errors.

Using the above as a guide, a weighted checklist as shown in table 5.4 was developed as a means for quantitatively evaluating how the LUT did in the test.

This gives a quantitative evaluation from individual functions, to grouped functions to the system as a whole. The quality factor for the major categories and the overall system as given will be 1 for a perfect validation and 0 for a totally unacceptable one. The weights for the categories and their constituent elements are assigned based on their importance. This may vary from one application to the other. The percentage success for a test element may be assigned based on the percentage of parameters or situations for which it showed the expected behaviour.

5.4 TEST SYSTEM CONFIGURATION AND ARCHITECTURE

5.4.1 Configuration

The testing is done incrementally as shown in fig. 5.2. First the protocol layers are tested, then a complete Teletex

Table 5.4 - TEST EVALUATION SCHEME: A Test Score Sheet

Major Category	Functions	Weight (max Score) Smax	Score			Comments
			Init. Si	Resp. Sr	Total St	
1. Connect.	<ul style="list-style-type: none"> - Parameter negotiation - (n-1)-layer addr. translation - Handling refusal - Handling collision 					
	$\text{Quality factor}_1, f_1$ $(\text{= } \sum St / \sum Smax)$					
2. Data	<ul style="list-style-type: none"> - Activity initiation - Activity continuation - Send/Receive data - Activity marks minor - Activity marks major - Normal activity end - Activity discard - Activity resync. 					
	$\text{Quality factor}_2, f_2$					
3. Release	<ul style="list-style-type: none"> - Normal - Abnormal 					
	$\text{Quality factor}_3, f_3$					
4. General	<ul style="list-style-type: none"> - Correct parameters - Dealing with inval. user requests - Handling provider errors - Handling provider failure - Handling peer errors 					
	$\text{Quality factor}_4, f_4$					
Final Quality factor, f ($= \sum f_i / \sum i, (i=1,2,\dots,4)$)

Fig. 5.2 - TEST SYSTEM CONFIGURATIONS

system is tested in a stand-alone and coupled systems mode. Stand-alone testing includes some amount of debugging and checks on the implementation. The system structures associated with the different configurations are shown in figure 5.3.

In the stand-alone configuration, only one TM is used for simplicity. For an n-layer LUT, its test script contains a related sequence of interactions for the n+1 and n-1 layer interfaces of the LUT. Thus, the TM plays the role of both user and provider with respect to the LUT. To make this possible, the TTX_process, at creation, is passed the TM_pipe as a representation of both the A_pipe and the N_pipe.

In the coupled systems testing configuration we are interested in determining the degree to which the systems satisfy the protocol standards. To simplify the encoding of the script, the test session and the result analysis, one of these Teletex systems is permanently configured as the initiator and the other, as the respondent.

Three levels of coupling can be identified for these two interworking Teletex systems:

- First, they can be made to run on the same terminal as two process groups (with a common ancestor) communicating with each other by UNIX pipes.

- Secondly, they could run on two separate terminals under the same host using operating system-supplied process coupling mechanisms, for instance, the communications functions of Berkeley 4.2 [4] could be used.

- Thirdly, a local Teletex system could be coupled to a remote Teletex system of the same or a different implementation via Datapac using the X.25 network board. This interworking mode is the ultimate goal of the system. The coupling here can be viewed in the same light as the last one described, the only distinction being that the inter-system linking is provided by X.25 hookup to Datapac instead of the UNIX operating system facilities.

Fig. 5.3 - Test System Structures

Note that in a test configuration involving a local and a remote TM (coupled systems), their test scripts will be closely related since what is sent from one side is basically what is received at the other.

5.4.2 Architecture

The overall test system structure has already been shown in figure 5.3. Figure 5.4 shows the structure of the TM_process that oversees to the conduct of the test. This process and the TTX_process exchange interactions via their input pipes. The initial input to the TM_process is a test script file as previously described. The TM_process uses this to generate interactions which it pipes to the TTX_pipe for the attention of the TTX_process. The TTX_process pipes interactions resulting from its activities to the TM_pipe. These are analyzed by the TM_process using the expected results components of its test script.

Some characteristics of this test structure are worthy of note. The TM_process in its dealing with the TTX_process plays the role of user and/or provider of the services of the latter. The layer under test, the n layer say, does not need to know if it is interacting with the $n+1$ and/or the $n-1$ layers, or with the TM. One TM (in the stand-alone configuration) can test both the $n+1$ and the $n-1$ interface functions simultaneously using just one test script file. The mapping of the i/o pipes for the system's processes is not fixed but configured at startup from supplied parameters. Details on this are given in the next section. Finally, the same TM_process is used for testing all protocol layers without modification because of the interaction-passing nature of the test scheme and the independence of the TM logic from that of the layer under test.

5.5 IMPLEMENTATION DETAILS

5.5.1 Test System Initiation and Operation

At startup, the TM_process is the parent process. It creates the TTX_pipe and any other pipes needed, depending on the desired configuration. The initialization information is passed to the TM_process at the console (under the UNIX shell)

Fig. 5.4 - Internals of the TM_PROCESS

as parameters of the execution command in the format:

```
run TM_process w x y z
```

where,

w is the test configuration identification;
x is the file containing the input test script;
y is the output file for the test results;
z is the file containing the entity to be tested.

w may be stand-alone, locally coupled or remotely coupled. In each case the TM configures the system suitably by creating the required processes (via fork calls) and also creating the input pipes required by these child processes. The processes and pipes required for different test system configurations have already been illustrated in figure 5.3. The child processes pick up their pipes ids, configure these properly for i/o and do an exec to their process code.

Note that for coupled systems, the startup is done independently for the initiator and the respondent systems (each with its own TM). The entity to be tested may be the transport layer or the session layer (with an already tested transport layer underneath). Some sample code for the testing can be found in appendix E.

5.5.2 Test Sequences

This section only describes the scheme used in obtaining test sequences for testing the Teletex implementation. Test sequence selection in general is a broad area and is outside the scope of this thesis. Nonetheless, interested readers can find work in this area in [40-42].

Here, the test sequences for LUT's test script are obtained from its FSM. To achieve a satisfactory amount of completeness, all state transitions in the FSM are traversed. For instance, the test sequence needed to check the transition from state 0.1 to state 0.2 of the calling side of the transport FSM (appendix C) is encoded as:

1. [SEND (T_CONN_REQ)]
[EXPECT (N_CONN_REQ)]
2. [SEND (status_request test_sdu)]
[EXPECT (status_response test_sdu (state=0.2))]

A sample test script encoding and test documentation for the Teletex T_layer initiator FSM (stand alone test configuration) is given in appendix D.

5.6. DISCUSSION

5.6.1 Evaluation of the Test Scheme

There are many merits to the test scheme just described:

It is very simple-minded and can be used to test any protocol system implementation. The TM_process is also very flexible in its interfacing to the LUT.

The TM's logic is independent of the logic of the layer being tested making it possible to use the same code for testing all layers. The simplicity of the TM's logic implies that it will be easy to comprehend code, debug and operate.

Since no restrictions were imposed on test script encoding any PDUs with any parameters can be encoded and tested. Furthermore, the method used for deriving the test script provides a logical relation between the implemented FSM and the testing. There is controllable testing since status can be forced.

The test scheme can be used for the different stages of testing, from debugging to the coupled systems testing.

Some disadvantages of the scheme are the following:

Encoding of the test script can be tedious since it is done by hand. But on the other hand, the mechanical nature of this activity implies that it is conducive to automation. Also, there is a dependence between the test script for the initiator and the respondent because of the dumb nature of the TM.

Another drawback of the scheme is the fact that for it to work to its full power, there is the need to define test SDUs and PDUs, backed up by a test FSM in the protocol layers. This

introduces a non-standard feature in the implementation and implies that though it can still be used to test implementations devoid of these test features, it may do so only in a limited capacity. For instance it could not be as effective for debugging since there will be no way to check the LUT's current state.

Another more general limitation of the testing (as with virtually all other tests) is that it is impossible to test all events and parameters in all states. Here, all events in all states were tested though not for all possible parameters. But for true completeness, all parameter values for all events for all possible PDUs in each state should be tested. The number of test sequences needed to satisfy this requirement is boundless because of the infinite combinations involved.

5.6.2 Other Test Approaches

Another common approach for protocol testing is the use of an intelligent TM that maintains a test FSM used for keeping track of the activities and status of the LUT [26]. For an n-layer testing, say, the logic for such a TM will be analogous to that of the n+1 layer with additional test logic incorporated.

Thus, such a test scheme is more difficult to design and implement. Consequently, the test software itself may not be totally correct since it can be quite complex. In such a situation, it will not be easy to determine if the blame for an unexpected behaviour should go to the LUT or, indeed, to the test software itself!

Furthermore, monitoring of the activities of the LUT will not be as straightforward as in the current example, and its interfacing to the LUT may not be as simple. The logic of the TM will be dependent on that of the LUT. Hence a separate TM software will have to be written for each layer to be tested.

An advantage of such a scheme may be its ability to make independent decisions on what actions to take depending on the status or the behaviour of the LUT, and, its ability to generate complex or random test sequences automatically.

In some protocol system implementations, a procedure-based approach as opposed to an interaction-based one is used [21,26]. In such cases, the test scheme presented can also be used, but only with some modifications. A decoder will have to be added to the TM's logic for interpreting the interactions obtained from the test script and deciding on the procedure to call in the LUT. The procedures to call and the parameters to use for the call may be suitably encoded in the test script. Thus we now have the drawback that the TM's logic may not be as independent from that of the layer to be tested.

5.7 CONCLUSION

Protocol testing is a multi-stage activity and any testing scheme adopted should be flexible enough to handle the many testing phases. An interaction-passing scheme based on a simple TM process driven by an input test script consisting of test actions and expected results has been presented. It has been shown that such a scheme is simple to understand, design, code and operate, and flexible in its interface to the LUT. Besides, the logic for the TM is independent from that of the LUT and the scheme is capable of handling all testing phases from debugging to the coupled systems stage.

Though this has not been implemented, an automatic test script generating tool will be a welcome addition to the test system described. This should then take the burden out of script encoding, making the scheme even more attractive.

CHAPTER. 6

DISCUSSION

6.1 MEETING DESIGN GOALS

From the elaborate descriptions of the UNIX-based system design, implementation and testing presented in the previous chapters, an evaluation can now be carried out on how well the design goals stated at the onset were satisfied. Some of these evaluations have already been presented in the associated chapters so this is mainly a recapitulation.

6.1.1 System Structure

On the question of portability, the evolved system architecture exhibits structural portability. It was necessary to design the UNIX-based Teletex system (session and transport layers) as a single Teletex process composed of a session and a transport layer pseudo-processes for UNIX-related performance reasons. However it was shown that portability benefits accrued from this approach since a single process structure is easy to port to other operating system with limited multitasking facilities like CP/M.

The pseudo-processes had an interface shell dealing with queuing/dequeuing of interactions with a configurable I/O ports mapping, and an interactions processing package for the associated layer's functions whose internals were further modularized into clear-cut protocol-related, testing and protocol-independent components. This modularization ensured that the pseudo-processes could be readily transformed into full-fledged processes if so desired as a result of, for instance, porting to an operating system with better multitasking facilities like ADA, RMX/86 etc. Then, only the interface shell will need modification, and the UNIX-pipes will become RMX/86 mailboxes, ADA buffer tasks and so on.

Details on the portability aspects of the system are covered in [33].

Protocol extensibility was achieved by adopting a table-driven FSM, since a table is easy to modify, coupled with the fact that an encoding scheme which lent itself mechanical updating was adopted. Thus the system could be upgraded in a step-wise fashion to handle ISO protocols.

6.1.2 Testing Approach

On the issue of testing, it was shown how an interaction-passing test scheme supported by protocol extension results in a test scheme with the following features:

- a simple and flexible interface as a direct consequence of using an interaction-based design for the test system;

- a controllable testing of the LUT because protocol extension in the LUT through the addition of test SDUs, PDUs and a test FSM made it possible to monitor the internal status of the LUT even though it was tested as a black box;

- a simple test logic resulting from the fact that all the LUT-bound interactions and the expected response from the LUT are encoded into a test script file: The test program simply 1) reads this file; 2) gets the next test interaction and the expected result from it; 3) sends the test interaction to the LUT; 4) receives the output interaction from the LUT and compares this to the expected result; 5) writes the output interactions and the result of the expected/actual interaction comparisons to an output file for analysis by the test conductor;

- a flexible and configurable usage of the test system because of the simplicity of the TM's logic, coupled with the interaction-passing scheme and protocol extension. Thus it was possible to use the test scheme for all phases of the testing, from debugging to remote testing.

6.2 OTHER APPROACHES

As explained earlier, the design presented here is based on interaction-passing, implying that the interface between

the system modules is via inter-module messaging. Such a design approach is said to be buffer-oriented since interaction-passing involves i/o-queues buffering of the interactions. The alternative design approach is procedure-oriented where the inter-module interfaces are procedures as opposed to interactions [29].

By nature, a procedure-oriented protocol system is a single-process system, with the control logic embedded in a main program, MAIN say, (application level). MAIN initiates processing by calling the next lower layer interface procedures. Specific procedures are called for specific functions. The functions may be requesting or accepting a connection, sending or receiving inter-system messages, etc. The calls are generally cascaded down from one layer to the other, all the way to the corresponding procedure at the bottom layer.

The current Teletex system could be adapted to a procedure-oriented design by transforming all the defined inter-module interactions into corresponding interface procedures for the session and transport layers. This has been done here as shown in figure 6.1. In this figure, some interactions have been grouped into just one interface procedure to enable the structure graph representation to fit a page. The call cascading mentioned above can be seen in this figure, for instance to receive data: S_DATA_IND gets called by MAIN then S_DATA_IND calls T_DATA_IND. Note that MAIN hangs up until the call gets propagated all the way down and returns all the way back to it.

A problem with the procedure-oriented design approach is the need for MAIN to poll for incoming interactions. This polling is needed since all lower layer modules are passive and cannot initiate autonomous processing actions. They get activated only when called from above. Polling overheads can be offset by including a check for incoming messages when sending outgoing ones. But another issue arises if this check is included. Should the outgoing interaction be sent before the incoming one is received, or vice-versa?

FIG. 6.1 - Procedure-oriented structure for the Teletex system

If the incoming message were first received, from the network level say, then the initial send call has to return all the way up through the intervening layers to MAIN. Then MAIN will call S_DATA_IND which results in calls propagating all the way down to the network level to receive the incoming message. When the incoming message eventually propagates up to MAIN, the latter retries sending the previously interrupted outgoing message.

This scheme has some undesirable characteristics. First, the send/receive procedures will have a complicated logic because of the cross-wise nature of the send/receive activities. Furthermore, if messages are received at a very high rate, there is the possibility of accumulation of outgoing messages because of the special priority given to incoming messages.

Alternatively, if the outgoing message is processed completely through before the incoming one is received, then the incoming message will not be received until the call returns and a separate receive call is initiated by MAIN. Thus, receipt of the incoming message is delayed by the amount of time required to send out the outgoing message, return to MAIN, and then call the receive procedures.

Thus, the first scheme favours incoming message while the other favours the outgoing ones.

The main problem with the procedure-oriented approach as described here is the fact that the protocol layers cannot autonomously receive incoming messages or initiate any other independent actions thus necessitating polling. Polling is an awkward and inefficient mechanism in terms of processing, and generally requires a relatively complicated logic compared to non-polling mechanisms. As was discussed in chapter 3, the layers of a full-duplex communication system require some level of autonomous processing. Using passive modules generally results in significant processing constraints and limitations.

Despite its other limitations, the procedure-oriented design satisfies most of the non-functional requirements

stated in chapter 3. Its single process structure makes for ease of portability. Extension can be accomplished by the addition of new interface procedures. It is also possible to reconfigure its layers.

6.3 MULTIPLE SESSIONS

Though the current implementation provides only one session connection to the user, the system can be easily configured to handle multiple simultaneous session, assuming that the network facilities can maintain multiple communication links. That the design could be easily modified to include this capability demonstrates its high level of configurability. Thus it becomes easy to extend the system to furnish this feature which is present in some enhanced versions of ISO's session protocol.

A multiple session capability can be added by using multiple copies of the TTX_process, with each process dealing with a specific session connection. This is done by modifying the application interface module of fig. 4.4 to the structure of fig. 6.2. The interface module now essentially plays the role of a session connection manager. Note that no extensive changes are needed for this extra capability:

Except for the extra requirement of being able to use a session id (S_id), the design remains essentially the same. Thus there are no structural changes. The only changes will be a minor addition to the code of the session pseudo-process' interface procedure, S_PI to make it add its S_id parameter to A_pipe-bound interactions; and adding the S_id parameter to the interface procedures definitions and codes.

6.4 OTHER ISSUES

6.4.1 UNIX Performance

Some enhanced versions of UNIX have features better suited to protocol system design thus overcoming some of the major deficiencies of standard UNIX. Notably, Berkeley 4.2 has provisions for non-blocking i/o and some inter-process memory sharing among other enhancements [4]. Thus the use of these

- (x) Kernel calls for creating the indicated entities. The A_pipe gets created once, on the first call to START_TTX only.
- (y) Kernel calls to kill a given TTX_process and delete its input pipe.

Fig. 6.2 - Possible Configuring of the Teletex System for multiple Sessions.

enhancements removes most of the buffering and data transfer limitations of standard UNIX described in chapter 4.

However, the use of enhanced UNIX makes it impossible to port the implementation to all versions of UNIX. Thus there is an issue of performance/portability trade-off.

6.4.2 Teletex FSMs

Some incompleteness or ambiguities were noted in the Teletex FSM details (refer to appendix C). Letting DS and DR refer to document send and receive FSM respectively, the following observations were made:

- No action is given for handling timeouts in DS and DR even though timers are started and stopped in many instances within them. The actions for handling timeouts given in the session calling and receiving side FSM are specifically described as not applicable for transitions from DS and DR. This omission was handled here by using the inactivity timeout actions given in the session FSMs for DS and DR.

- The STOP TIMER action given in DS1 after transitions from DS8 and DS9 are superfluous since the timer has already been stopped in DS7 and are not shown as having been started anywhere else in-between.

- The transition out of DR5.1 to DR5 (on the way to DR1) should be the result of an S-SYNC MAJ RESP POS event which is not shown.

- In the transition from DR3 to DR5, R:CSUI/CDPB should be R:CSUI/CDE. This can in part be noted from the fact that CDE is sent from DS but there is no mention of it being received at DR nor any associated action given. Also, analysis shows that DR5 is the state where DR completes the acknowledgement of received pages at the end of a document.

- The transition from DR3 to DR4 should read R:CSUI/CDPBP(I) AND (P-Q)=(W-1) AND I=P, instead of simply R:CSUI/CDPB(I), because DR4 is the state where DR demands

minor synchronization points response from the user when its receive window gets full.

- In DRw (shown with transitions to DR7), it is ambiguous how the user gets notified of detected error. Further, S-ERROR IND is shown as if it were an event from the user when in fact it should be an indication to the user. Here it is assumed that the user is notified of the errors by DR, and the latter only starts resynchronization or error recovery actions after it obtains the associated requests from the user.

CHAPTER 7

CONCLUSION

7.1. DESIGN REFLECTIONS

Despite some of its deficiencies in dealing with the time-critical activities associated with communications software, standard UNIX can be used to design a protocol system with an acceptable level of performance if some simple rules are observed. The rules include limiting the amount of piping and the number of protocol-related processes in the system with an aim to reducing kernel i/o calls and data transfer overheads. It was established that observing these rules result in a system architecture - in this case a single Teletex process with a transport and a session level pseudo-processes - which is easy to port to other operating system environments since the less the overall number of processes, the easier porting becomes to operating systems with limited multiprocessing facilities.

An extensive analysis of the Teletex protocol was carried out. It was shown that Teletex is ISO-upwards extensible since the functionality of the Teletex S.62 session/document is hierarchically embodied in ISO's basic session protocol and the Teletex protocol is analogous to ISO transport class 0.

Before a protocol system implementation is undertaken, it is necessary to undertake an extensive analysis of its mechanisms to establish the non-functional system requirements and map out critical data flow patterns. A novel approach to carrying out this analysis was given. It was based on special data flow graphs showing the physical flow patterns of data units between the adjacent protocol layers, and the logical data flows patterns between peer entities in different systems. Such a data flow structure, it was shown, provides an instant high level view of the protocol/service interworkings

and help in demarcating the interfaces between the system's protocol layers, and establishing the level of sophistication required of the implementation.

It was also shown how a configurable protocol system could be obtained by cleanly separating protocol-dependent and protocol-independent elements and functions. This was achieved for the Teletex system by a hierarchical modularization of the Teletex system architecture inwards.

- first, the Teletex process was demarcated into a session and a transport level pseudo-processes;

- second, the pseudo-processes were cleanly demarcated into protocol-dependent components associated with the protocol service functions for the layer in question, and protocol-independent components exclusively devoted to interacting with the pseudo-processes surroundings; and,

- third, the protocol/service-related components were further modularized into protocol and testing sub-components with local and end-to-end constituents.

A flexible and configurable test scheme for the Teletex system was also presented. This test scheme was based on interaction-passing between the testing program and the layer under test in an excitation/response format. Simplicity was achieved by encoding all the test-interaction/expected-interaction pairs in a test script file which was used by the testing program to generate test excitations for the layer under test and analyze the consequent responses. The above features resulted in a simple test conductor-to-TM and TM-to-LUT interfaces. It was shown that by extending the protocol of the layer under test to include test SDUs, PDUs and a corresponding test FSM, it was possible to: 1) test this layer as a black box and at the same time, be able to keep track of its internal status by using status test functions; 2) make the testing more controllable and shorten the test sequences since specific protocol-related actions could be requested of the layer under test.

These features of the test approach, coupled with an automatic configuring capability of the test structure by the Test_Manager process, made it possible to use the same test

and help in demarcating the interfaces between the system's protocol layers, and establishing the level of sophistication required of the implementation.

It was also shown how a configurable protocol system could be obtained by cleanly separating protocol-dependent and protocol-independent elements and functions. This was achieved for the Teletex system by a hierarchical modularization of the Teletex system architecture inwards:

- first, the Teletex process was demarcated into a session and a transport level pseudo-processes;
- second, the pseudo-processes were cleanly demarcated into protocol-dependent components associated with the protocol/service functions for the layer in question, and protocol-independent components exclusively devoted to interacting with the pseudo-processes surroundings; and,
- third, the protocol/service-related components were further modularized into protocol and testing sub-components with local and end-to-end constituents.

A flexible and configurable test scheme for the Teletex system was also presented. This test scheme was based on interaction-passing between the testing program and the layer under test in an excitation/response format. Simplicity was achieved by encoding all the test-interaction/expected-interaction pairs in a test script file which was used by the testing program to generate test excitations for the layer under test and analyze the consequent responses. The above features resulted in a simple test conductor-to-TM and TM-to-LUT interfaces. It was shown that by extending the protocol of the layer under test to include test SDUs, PDUs and a corresponding test FSM, it was possible to: 1) test this layer as a black box and at the same time, be able to keep track of its internal status by using status test functions; 2) make the testing more controllable and shorten the test sequences since specific protocol-related actions could be requested of the layer under test.

These features of the test approach, coupled with an automatic configuring capability of the test structure by the Test_Manager process, made it possible to use the same test

code for all phases of testing, from debugging to coupled systems testing.

7.2 STATUS OF PROJECT

A complete data flow analysis of the Teletex system has been carried out and a corresponding detailed structure of the Teletex system has been obtained. The transport layer pseudo-process service package, T_Service, has been fully coded by another project participant. The author has coded the enveloping shells comprising the MAIN logic of the Teletex_PROCESS and of its pseudo-processes' interface procedures for interacting with the surroundings. The service package has been linked with this enveloping shell.

Also, the author has fully coded and compiled the test system software. The test software has been used to test the T_Service/envelopes software described above. This testing, of the transport layer implementation has been fully carried out in the stand alone system configuration. A configuring program for connecting two Teletex systems locally for testing has been written and compiled and coupled-systems testing should commence shortly.

The author has started work on encoding the session layer service package, S_Service. All the FSM tables have already been encoded including the action procedures. What remains to be done is writing the procedures to encode/decode session-level interactions. When this is completed, session layer testing should commence.

7.3 FURTHER WORK

It is required to complete the locally coupled testing of the transport layer and go on to test two Datapac-connected transport layers.

Completion of the coding of the session layer software is also desired. This activity will proceed in parallel with the transport layer testing, and testing of the session layer software from a stand-alone to a remotely coupled configuration will commence once the software is fully coded. The already tested transport layer will be used underneath the session for coupled-systems testing.

APPENDIX C

TELETEX FSMs

These are reproductions from reference [16] of the Bibliography. The following protocol FSMs are given:

a) Session:

- Calling Side;
- Called Side.

b) Document:

- Sending Side;
- Receiving Side.

c) Transport:

- Calling Side;
- Called Side.

So far, no attention has been paid to the nature of the application/presentation level processes that interact with the Teletex gateway process. Eventually, it will be necessary to choose a suitable application/presentation protocol compatible with Teletex, and then design and implement it. The presentation level functions will deal with Teletex encryption and text formatting into the format specified by the associated Teletex recommendations. The application level functions will be required to provide suitable interfaces (e.g. file transfer facilities) to the end-user, and also provide interfaces by which other Teletex-independent processes within the local area network could use the services of the Teletex gateway.

On the issue of testing, a possible activity is the design of an automatic test script encoding module which will do away with the drudgery of manual script encoding. It is also necessary to develop more formal methods for qualitatively and quantitatively analyzing the test results, possibly, automatically.

Further on the horizon, it may be necessary to attempt some example porting of the Teletex system to other operating systems like ADA, RMX/86 and CP/M.

Performance was of secondary consideration in this design although some amount of performance analysis was done by one of the project participants [33]. However, it will be helpful to carry out a complete performance analysis of the Teletex system to evolve ways of maximizing its efficiency while preserving its modular structure. It may also be interesting to compare the performance of this interaction-based system to procedure-oriented designs in order to obtain a quantitative evaluation of the two approaches side by side.

BIBLIOGRAPHY

1. Christian, K., The UNIX Operating System, Wiley-Interscience, 1983.
2. Bourne, S.R., The UNIX System, Addison-Wesley, 1982.
3. Kernighan, B.W., Ritchie, "UNIX Programming", Second Edition (May 11, 1983), SUN UNIX System Manual, Programming Tools, SUN Microsystems Inc.
4. Joy, W., Cooper, E., Fabry, R., Leffler, S., McKusick, K., Mosher, D., "4.2 BSD System Manual", Draft of September 1, 1982, University of California, Berkeley.
5. Kernighan, B.W., Ritchie, D.M., The C Programming Language, Prentice-Hall, 1978.
6. ISO DP 7498, "Reference Model for Open Systems Interconnection", February 1982.
7. ISO/TC 97/SC16, "Basic Connection Oriented Session Service Definition", March 1983.
8. ISO/TC 97/SC16, "Basic Connection Oriented Session Protocol Specification", March 1983.
9. ISO/TC 97/SC 16 N 1169, "Information Processing Systems - Open Systems Interconnection - Transport Protocol Specification", Tokyo, June 1982.
10. CCITT Recommendation, "Teletex Service", Vol II, Fascile II.4, Rec. F.200.
11. CCITT Recommendation, "Terminal Equipment Use in the Teletex Service", Vol II, Fascile VII.2, Rec. S.60.
12. CCITT Recommendation, "Character Repertoire and Coded Character Sets for the International Teletex Service", Vol. II, Fascicle VII.2, Rec. S.61.

13. CCITT Recommendation, "Control Procedures for the Teletex Service", Vol. VII, Fascicle VII.2, Rec S.62.
14. CCITT Recommendation, "Network-Independent Basic Transport Service for Teletex", Vol VII, Fascicle VII.2, Rec. S.70.
15. CCITT Recommendation X.25,
Document APVII Number 7-E, June 1980.
16. Teletex, TCTS Terminal Specification, 1982 10 13, Release 3.0.
17. Piatkowski, T.F., "A Formal Definition of the ISO-ANSI Open Systems Interconnection Reference Model - A Brief Overview and Proposal for a State-Oriented Systems Approach,
Proceedings, 1st International Conference on Distributed Computing Systems, October 1979, pp. 79-94.
18. Knightson, K.G., "The Transport Layer",
Computer Communications Review, October 1982.
19. Sevcik, P.J., Nichols, P.J., "Packet Transport Design Considerations",
Proceedings, Computer Networking Symposium, 1978, pp. 129-137.
20. Nelson, J., "Implementation of Encryption in an Open System Architecture",
Proceedings, Computer Networking Symposium, 1979, pp. 198-205.
21. Buhr, R.J.A., Mackinnon, D.A., "Evolving Toward Open System Interconnection",
INFOCOM 83, San Diego, April 1983.
22. Buhr, R.J.A., "Protocol Development for Open System Interconnection: Research Plans for Carleton's Local Area Network",
Report SCE-83-6, Department of Systems and Computer Engineering, Carleton University.
23. Piatkowski, T., "Remarks on ADCCP Validation and Testing Techniques",
NBS Trends and Applications Symposium, May 29, 1980.

24. Bartlett, K., Rayner, D., "The Certification of Data Communication Protocols",
NBS Trends and Applications Conference, May 29, 1980, pp. 12-17.
25. Bochmann, G.V., Cerny, E., "Protocol Assessment",
A report prepared for Dendronics Decisions Limited,
Montral, Quebec, Feb 3 1983.
26. Bochmann, G.V., Cerny, E., "Implementation Test Plan for the BIG File Transfer Protocol",
CERBO Informatique Inc., Westmount, Quebec, Jan. 1983.
27. Pouzin, L., Zimmerman, H., "A Tutorial on Protocols",
Proceedings of the IEEE, Nov. 1978, pp. 1346-1370.
28. Tanenbaum, A.S., Computer Networks,
Prentice-Hall, 1981.
29. Lauer, H.C., Needham, R.M., "On the Duality of Operating System Structures",
Proceedings, Second International Symposium on Operating Systems, IRIA, Oct. 1978, reprinted in Operating Systems Review, 13,2 April 1979, pp. 3-19.
30. Bowen, B.A., Buhr, R.J.A., The Logical Design of Multiple Microprocessor Systems,
Prentice-Hall, 1980.
31. Buhr, R.J.A., System Design with ADA,
Course notes, Department of Systems and Computer Engineering, Carleton University,
(To be published by Prentice-Hall).
32. Bowen, B.A., Brown, W.R., VLSI Systems Design for Digital Signal Processing, Vol II: Systems Design,
Course notes, Department of System and Computer Engineering, Carleton University,
(To be published by Prentice-Hall).
33. Griffiths, C.G., Protocol Software Portability,
Thesis - Carleton University, 1983,
(To be published).

34. ISO/TC97/SC16, "Test Methodology for OSI Protocols", Canada, 1983.
35. Anart, J.P., "A Protocol Independent System for Testing Protocol Implementations",
Project RHIN, ADI, Tour Fiat Cedex 16, 92084
PARIS-LA-DEFENCE.
36. Ural, H., Probert, R.L., "Architecture for testing Protocol Implementations",
Proc. of CIPS Conference 1983, Ottawa, May 1983.
37. Ansart, J.P., "Test and Certification of Protocols",
Protocol Testing - Towards Proof? Vol. 2: Testing and Certification, (ed. D. Rayner et. al.), NPL, 1981, pp. 119-126.
38. Ansart, J.P., Damidau, J., "CERBERE: A Tool to Keep an Eye on High Level Protocols",
Protocol Specification, Testing, and Verification, (ed. C. Sunshine), North-Holland, 1982, pp. 171-185.
39. Nightingale, J.S., "Protocol Testing Using a Reference Implementation",
Proc. IPIP WG6.1 2nd Int. Workshop on Protocol Specification, Testing and Verification (Idyllwild, CA, May 17-20, 1982) North-Holland, 1982, pp. 513-522.
40. Ural, H., Probert, R.L., "User-Guided Test Sequence Generation",
Proc. of 3rd IPIP Workshop on Protocol Specification Testing and Verification, Zurich, May 1983.
41. Hanford, K.V., "Automatic generation of Test Cases",
IBM Systems Journal, Vol. 9, NO. 4, 1970, pp. 242-257.
42. Sarikaya, B., Bochmann, G.V., "Some Experience with Test Sequence Generation for Protocols",
Protocol Specification, Testing, and Verification, (ed. C. Sunshine), North-Holland, 1982, pp. 555-567.

GLOSSARY OF TERMS

OSI	Open Systems Interconnection
ISO	International Standards Organization
CCITT	International Telegraph and Telephone Consultative Committee
TCTS	TransCanada Telephone System
TTX	Teletex
PDU	Protocol Data Unit
SDU	Service Data Unit
A, P, S, T, N	stand for Application, Presentation, Session, Transport and Network respectively
SS, TS, NS	stand for Session Service, Transport Service and Network Service respectively
interaction	an item of work exchanged by the system modules and constituting a logical interface between them
ACT, MGT	activity, management
activity	a logical piece of work performed during a session
token management	a function used to determine which side of two interworking protocol entities has the right to exercise certain functions, e.g. send data PDUs
segmentation/concatenation	sometimes referred to as segmenting/reassembly is a protocol function that maps one SDU into multiple PDUs; the destination reassembles the segmented PDUs
OWC, TWA, TWS	Teletex duplex modes: One Way Communication, Two Way Alternate, Two Way Simultaneous

document	an ordered set of one or more pages delivered to their destination(s) as a single entity.
page	basic data unit of the Teletex service (roughly equal to a standard office page)
synchronization mark	a numbered mark inserted in the text stream for synchronization and error recovery purposes
checkpoint	synchronization mark
source/sink	possessor/non-possessor of the data transmit token
acknowledgement windows	maximum number of sequentially numbered Teletex pages that the transmitter can send without receiving an acknowledgement from the receiver
pipe	the UNIX operating system's interprocess messaging channel
fork	UNIX system call that spawns a child process for the calling process
exec	UNIX system call that overlays the calling process with a new program
FSM	Finite State Machine
i/o	input/output
TM	Test Manager - module overseeing to the system's testing
LUT	Layer Under Test
T_PI, S_PI, n_PI	interface procedures for the Transport, Session and n layer pseudo-process respectively (stand for x_Process_Interaction)

APPENDIX A

GRAPHICAL NOTATIONS

- An uncommitted module.

- A module of procedures and data structures (passive ADA-like package)

- A process

- A UNIX pipe

- A data structure

- An access arrow

- A data flow element

- The dot indicates possible on access to entity D. (Blocking occurs when it is required to wait on some condition to become true).

- The circular arrow is used for ordering access arrows.

SOME EXAMPLES ON USING THE GRAPHICAL NOTATIONS:

- Data flow element a flows from uncommitted module A to B, while b flows from B to A.

- Interface procedure c1 of package C is always called first before a call can be made to c2.

- Process P1 writes data item q1 to pipe Q. Process P2 reads data item q2 from Q; it gets blocked on Q when there is nothing to receive.
- Process P2 contains a module m with an interface procedure m accesses a data structure n; it reads a from it and writes b to it.

APPENDIX B

TELETEX PDU ABBREVIATIONS

Abbreviation

Meaning

S.62 Session PDUs:

CSS	Command Session Start
RSSPN	Response Session Start Positive
RSSN	Response Session Start Negative
CSE	Command Session End
RSEP	Response Session End Positive
CSA	Command Session Abort
RSAP	Response Session Abort Positive
CSUI	Command Session User Information
RSUI	Response Session User Information
CSCC	Command Session Change Control
RSCCP	Response Session Change Control Positive

S.62 Document PDUs:

CDS	Command Document Start
CDC	Command Document Continue
CDCL	Command Document Capability List
RDCLP	Response Document Capability List Positive
CDE	Command Document End
RDEP	Response Document End Positive
CDD	Command Document Discard
RDDP	Response Document Discard Positive
CDR	Command Document Resynchronize
RDRP	Response Document Resynchronize Positive
CDUI	Command Document User Information
RDGR	Response Document General Reject
CDPB	Command Document Page Boundary
RDPBP	Response Document Page Boundary Positive
RDPBN	Response Document Page Boundary Negative

S.70 Transport PDUs:

TCR	Transport Connection Request
TCA	Transport Connection Accept
TCC	Transport Connection Clear
TDT	Transport Data
TBR	Transport Block Reject

APPENDIX C

TELETEX FSMs

These are reproductions from reference [16] of the Bibliography. The following protocol FSMs are given:

a) Session:

- Calling Side;
- Called Side.

b) Document:

- Sending Side;
- Receiving Side.

c) Transport:

- Calling Side;
- Called Side.

c-1

TELETEX DOCUMENT STATE TRANSITION DIAGRAM-INTERNAL STATE DETAILS
(SENDING PROTOCOL)

(Reproduced from [16])

TELETEX TRANSPORT STATE TRANSITION DIAGRAM - INTERNAL
(CALLED SIDE) STATE DETAILS

C-6

(Reproduced from [16])

APPENDIX D

DOCUMENTATION FOR TRANSPORT LAYER SOFTWARE TESTING

1. Test Control

These test interactions are used to test the transport layer software in a stand alone configuration and thus, can be viewed as incorporating some debugging functionality.

There is a single test controlling process, the TEST_MANAGER (TM) process. This sends interactions to the layer under test (LUT) via the TTX_pipe. For this phase of testing, this process manages both the session-transport and the transport-network interfaces. This is possible since the process is essentially dumb and just reads a test script from a coded test script file. The script includes the interaction to be sent, and what to expect in reply. As a result session and network level interactions can be inter-mixed in this test script, making it possible for only one process to manage the test processing at this stage.

On the other hand, the LUT sends both the application and network level interactions to the same pipe, the TTX_pipe. This is so because the pipe file descriptors for network and transport level interactions which get passed to the TTX_process (which controls the LUT) when it is created are one and the same.

2. Test Description

Two types of tests have been identified which are:

- Action-check test
- State-check test (St)

In action-check tests, the LUT is given a specific input and its reaction checked against the standard reaction specified in the protocol. The action may be

- To send a TPDU to its peer protocol layer via the N interface (T-T);
- To send a TSDU to its user, the session level (T-S);
- To send a command NSDU to its provider, the network level (T-N).

In the following test description, these test types will be distinguished whenever possible.

3. Test Encoding

The following test encoding scheme is used for the test script:

a) Interactions are coded in the form:

```
sent interaction : expected interaction :
sent interaction : expected interaction: ...
etc.
```

where individual interactions are separated by a colon ':'. If nothing is to be sent or expected at any point this is indicated by a null, with nothing between the two colons concerned, that is '::'.

Though the script is assumed to consist of characters, integers can be represented by a three digit ascii code preceded by a backslash '\'. Carriage returns are ignored and can be included in the script by using its ascii representation '13'. The special characters '\' and ':' can also be included in the script as '\\\' and '\:'. The special characters '\n' and '\r' can also be included in the script as '\\n' and '\\r'.

4. Test Sequences for the T-layer Caller FSM

The following are the test sequences for the Transport layer software in a stand alone configuration as derived from the Transport layer "Caller" FSM of the Teletex protocol:

TEST SEQUENCES

Test No.	Resp. Type	Session-Transport Interface		Transport-Network Interface		Comments
		Sent TSDU	Expected TSDU	Sent TSDU	Expected TSDU	
1	T-N	T_CONN REQ			NL_CONN REQ	
2	St	T_STAT REQ	T_STAT RES			state = 0.2
3	T-T			NL_CONN CONF+	NL_DATA REQ (TCR pdu)	
4	St	T_STAT REQ	T_STAT RES			state = 1.1
5	T-S		T_CONN CONF+	NL_DATA IND (TCR pdu)		+ve confirm
6	St	T_STAT REQ	T_STAT RES			state = 2.1
7	T-S		T_T_CRD IND	NL_NCRD IND		up credit
8	T-T	T_DATA REQ			NL_DATA REQ (TDT pdu)	outgoing data

9	St	T_STAT REQ	T_STAT RES			state = 2.1
10	T-N	T_S_CRD RES			N_T_CRD RES	down credit
11	T-S		T_DATA IND	N_DATA IND (TDT pdu)		incoming data
12	T-S		T_T_CRD IND	N_N_CRD IND		up credit
13	T-T	T_DATA REQ			N_DATA REQ (TDT pdu)	outgoing data
14	St	T_STAT REQ	T_STAT RES			state = 2.1
15	T-N	T_S_CRD RES			N_T_CRD RES	down credit
16	T-S		T_DATA IND	N_DATA IND (TDT pdu)		incoming data
17	T-S		T_T_CRD IND	N_N_CRD IND		up credit
18	T-T	T_DATA REQ			N_DATA REQ (TDT pdu)	outgoing data
19	St	T_STAT REQ	T_STAT RES			state = 2.1
20	T-N	T_S_CRD RES			N_T_CRD RES	down credit
21	T-S		T_DATA IND	N_DATA IND (TDT pdu)		incoming data
22	T-S		T_T_CRD IND	N_N_CRD IND		up credit
23	T-T	T_DATA REQ			N_DATA REQ (TDT pdu)	outgoing data
24	St	T_STAT REQ	T_STAT RES			state = 2.1
25	T-N	T_S_CRD RES			N_T_CRD RES	down credit
26	T-S		T_DATA IND	N_DATA IND (TDT pdu)		incoming data
27	T-S		T_T_CRD IND	N_N_CRD IND		up credit
28	T-T	T_DATA REQ			N_DATA REQ (TDT pdu)	outgoing data
29	St	T_STAT REQ	T_STAT RES			state = 2.1
30	T-N	T_S_CRD RES			N_T_CRD RES	down credit
31	T-S		T_DATA IND	N_DATA IND (TDT pdu)		incoming data
32	T-S		T_P_ERR IND	N_P_ERR IND		
33	T-N	T_DISC REQ			N_DISC REQ	

35	St	T_STAT REQ	T_STAT RES			state = 0.1
36	St	T_FORCE STATE REQ				state = 0.2
37	T-S		T_CONN CONF-	N_CONN CONF-		-ve confirms
38	St	T_STAT REQ	T_STAT RES			state = 0.1
39	St	T_FORCE STATE REQ				state = 0.2
40	T-S		T_CONN CONF-	N_LP_ERR IND		
41	St	T_STAT REQ	T_STAT RES			state = 0.1
42	St	T_FORCE STATE REQ				state = 1.1
43	T-S	T_DISC REQ			N_DISC REQ	
44	St	T_FORCE STATE REQ				state = 1.1
45	T-S		T_DISC IND	N_DATA IND (TBR pdu)		
46	T-N				N_DISC REQ	
47	St	T_FORCE STATE REQ				state = 1.1
48	T-S		T_DISC IND	N_LP_ERR IND		
49	T-N				N_DISC REQ	
50	St	T_FORCE STATE REQ				state = 0.2
51	T-S	T_DISC REQ			N_DISC REQ	
52	St	T_FORCE STATE REQ				state = 1.1
53	T-S	T_DISC REQ			N_DISC REQ	
54	St	T_FORCE STATE REQ				state = 2.1
55	T-S	T_DISC REQ			N_DISC REQ	
56	St	T_FORCE STATE REQ				state = 0.2
57	T-T			N_DATA IND (inval. pdu)	N_DATA REQ (TBR pdu)	unknown pdu sent

58	St	T_STAT REQ	T_STAT RES			state = 0.3
59	T-S		T_DISC IND	N_DISC IND		
60	St	T_STAT REQ	T_STAT RES			state = 0.1
61	St	T_FORCE_ STATE REQ				state = 2.1
62	T-T			N_DATA IND (inval. pdu)	N_DATA REQ (TBR pdu)	
63	St	T_STAT REQ	T_STAT RES			state = 0.3
64	T-S		T_DISC IND	N_DISC IND		
65	St	T_STAT REQ	T_STAT RES			state = 0.1
66	St	T_FORCE_ STATE REQ				state = 1.1
67	T-T			N_DATA IND (TCR pdu)	N_DATA REQ (TBR pdu)	
68	St	T_STAT REQ	T_STAT RES			state = 0.3
69	T-S		T_DISC IND	N_DISC IND		
70	St	T_FORCE_ STATE REQ				state = 1.1
71	T-S		T_DISC IND	N_DISC IND		
72	St	T_STAT REQ	T_STAT RES			state = 0.1
73	St	T_FORCE_ STATE REQ				state = 2.1
74	T-S		T_DISC IND	N_DISC IND		
75	St	T_STAT REQ	T_STAT RES			state = 0.1
76	St	T_FORCE_ STATE REQ				state = 2.1
77	T-S		T_DISC IND	N_DISC IND		
78	St	T_STAT REQ	T_STAT RES			state = 0.1

APPENDIX E

SAMPLE OF THE TEST CODE

Major Contents:

- a) C language extension definitions.
- b) TTX_TM.c -- The test module containing the main logic of the test process, "main(...)".
- c) LUT_driver.c -- The encapsulating shell of the protocol layer under test containing: i) the main logic of the Teletex process, "main(...)"; ii) the interface procedure of the Transport layer pseudo-process, "T_PI(...)".

```

/*----- C_extensions.h -----*/
/*                                     */
/*      C language extensions      */
/*-----*/

```

```

#define PROCEDURE
#define FUNCTION
#define IF if
#define ENDIF )
#define THEN

```

```

#define BEGIN (
#define ENDIF )
#define ELSE else
#define ENDELSE )
#define END )

```

```

#define WHILE while
#define ENDWHILE )

```

```

#define LOOP while(1)
#define ENDOLOOP )
#define F R for
#define ENDFOR )

```

```

#define CASE switch
#define OF (
#define ENDOF )
#define ENDCASE )
#define EXIT break
#define OTHERWISE default

```

```

#define RECORD struct
#define ENDRECORD )
#define INTEGER int
#define CHARACTER char

```

```

#define BOOLEAN int
#define IS ==
#define ISNOT !=
#define NOT !
#define AND &&
#define OR ||
#define MOD %

```

```

#define TRUE 1
#define FALSE 0

```

```

#define WRITE printf
#define WRITELN printf("\n")
#define READ scanf

```

```

typedef char *STRING;
typedef char *CHAR_POINTER;
typedef int *INT_POINTER;

```

```
typedef FILE *FILE_POINTER;
```

```
/*----- TTX_declare.h -----*/  
/*  
/*      CONSTANTS  DEFINITION      */  
/*-----*/
```

```
#define INTERACTION_SIZE 128  
#define TRACE_FILE_SIZE 4096  
#define UP 1  
#define DOWN -1  
#define BOTH 2
```

```

/*----- TTX_TM.c -----*/
/*
/* actions:
/*   - opens input test script file
/*   - creates pipes and the TTX process
/*   - processes test script and writes result to a
/*     specified output file..
/*-----*/

```

```

#include <stdio.h>
#include "TTX_declare.h"
#include "C_extensions.h"

```

```

FILE_POINTER    fp_in, fp_out;    /* in and out file pointers */
INTEGER         trace_inx;        /* indexes into trace buffer */

```

```
main(argc, argv)
```

```

INTEGER         argc;
CHAR_POINTER    argv[]; /* argv[0] is command name (not used here) */
                  /* argv[1] is input file
                  /* argv[2] is output file
                  /* argv[3] is file name of program to be tested */

```

```
BEGIN
```

```
FILE_POINTER    fopen(); /* script files descriptors */
```

```
BOOLEAN        done, test_get();
```

```

CHARACTER       rcv_buf[INTERACTION_SIZE+1],
exec_arg[7], /* arguments for child process exec call */
ascii_str[4], /* storage for ascii number representation */
sdu_1[INTERACTION_SIZE+2], sdu_2[INTERACTION_SIZE+2];
                  /* sent and expected interactions */

```

```

INTEGER         childpid, test_inx, i, j,
pipe_fd[2][2], /* pipes file descriptors */
TM_fd, TTX_fd;

```

```
/* open input and output files: */
```

```

IF (argc ISNOT 4) THEN
    error("*** TM - Invalid argument count\n");
fp_in = fopen( argv[1], "r");
IF (fp_in IS NULL) THEN BEGIN
    WRITE("*** TM - can't open %s\n", *argv);
    error("");
ENDIF

```

```

fp_out = fopen( argv[2], "w");
IF (fp_out IS NULL) THEN BEGIN

```

```

WRITE("*** TM - can't open %s\n", *argv);
error("");
ENDIF

/* create pipes: */

IF (( pipe( &pipe_fd[0][0]) IS -1) OR ( pipe( &pipe_fd[1][0]) IS -1))
    THEN error("*** TM - pipe creation failure\n");

/* spawn new process and send it to its process code: */

childpid = fork();
IF ( childpid IS -1) THEN
    error("*** TM - process creation failure\n");

IF ( childpid IS 0) THEN BEGIN
    exec_arg[0] = pipe_fd[0][0];
    exec_arg[1] = pipe_fd[0][1];
    exec_arg[2] = pipe_fd[1][0];
    exec_arg[3] = pipe_fd[1][1];
    exec_arg[4] = pipe_fd[0][0];
    exec_arg[5] = pipe_fd[0][1];
    exec_arg[6] = ' ';

    execl( argv[3], "T_test",exec_arg,0);
    error( "*** LUT - exec to LUT code unsuccessful\n");
ENDIF

/* establish the proper pipe configuration for TM process */

close( pipe_fd[0][1]); /* close write end of TM pipe */
close( pipe_fd[1][0]); /* close read end of TTX_pipe */
TM_fd = pipe_fd[0][0];
TTX_fd = pipe_fd[1][1];

/* test script interactions testing begins: */

done = FALSE;
trace_inx = 1;
test_inx = 1;
ascii_str[0] = '\';
trace_inx = 0;

WHILE (done IS FALSE) BEGIN
    done = (test_get(fp_in, sdu_1) OR test_get(fp_in, sdu_2));
    /* get sent and expected interactions from test script and
    /* set end of script flag if necessary */

    IF (sdu_1[0] ISNOT 0) THEN
        /* write to TTX_pipe: */
        IF (write( TTX_fd, &sdu_1[1], INTERACTION_SIZE) ISNOT
            INTERACTION_SIZE)
            THEN error("*** TM - TTX pipe write error\n");

    IF (sdu_2[0] ISNOT 0) THEN BEGIN

```

```

/* receive input interaction: */
IF (read( TM_fd, rcv_buf, INTERACTION_SIZE) ISNOT
    INTERACTION_SIZE)
    THEN error("*** TM - TM pipe read error\n");

/* compare received and expected interaction: */
IF (strcmp(rcv_buf, &sdu_2[1]) ISNOT 0) THEN BEGIN
    WRITE("*** ERROR -- unmatched response for test %d\n",
        test_inx);
    rewind_check( argv[2]);
    putc('*', fp_out);

ENDIF
rcv_buf[sdu_2[0]] = '0';
FOR (i=0; i<=sdu_2[0]; i=i+1) BEGIN

    ascii_encode( rcv_buf[i], &ascii_str[i]);
    FOR (j=0; j<4; j=j+1) BEGIN
        rewind_check( argv[2]);
        putc( ascii_str[j], fp_out);
    ENDFOR
ENDFOR

ENDIF
test_inx = test_inx+1;
ENDWHILE

fclose(fp_in); /* close input file */
fclose(fp_out); /* close output file */
kill(childpid, 9);

END

```

```
/* file rewinding procedure */
```

```
PROCEDURE rewind_check( name)
```

```
    STRING name; /* file name */
```

```
    BEGIN
```

```
        trace_inx = (trace_inx + 1) MOD TRACE_FILE_SIZE;
```

```
        IF ( trace_inx IS 0) THEN BEGIN
```

```
            fclose( fp_out);
```

```
            fp_out = fopen( name);
```

```
        ENDIF
```

```
    END
```

```
#define ESC '\'
```

```
#define TERMINATOR ':'
```

```
/* procedure for obtaining the next 2 test elements from the test script */
/* - the next test script is returned via buffer pointer 'test' */
```



```
/* - the 1st loc of the buffer contains the effective interaction size */
```

```
FUNCTION BOOLEAN test_get( fp, test)
```

```
FILE_POINTER    fp;    /* file pointer of input test script */
STRING          test;  /* pointers to the obtained interactions */
```

```
BEGIN
```

```
INTEGER i, c, c2, c3;
```

```
  i = 0;
```

```
  c = getc( fp);
```

```
  WHILE ((c ISNOT EOF) AND (c ISNOT TERMINATOR)) BEGIN
```

```
    IF ( c ISNOT '0') THEN BEGIN
```

```
      i = i + 1;
```

```
      IF ( c ISNOT ESC) THEN
```

```
        test[i] = c;
```

```
      ELSE BEGIN
```

```
        c = getc( fp);
```

```
        WHILE (c IS '0')
```

```
          c = getc( fp);
```

```
          IF (c >= '0' AND c <= '9') THEN BEGIN
```

```
            c2 = getc( fp);
```

```
            WHILE ( c2 IS '0')
```

```
              c2 = getc( fp);
```

```
            c3 = getc( fp);
```

```
            WHILE ( c3 IS '0')
```

```
              c3 = getc( fp);
```

```
            IF (c2='0' AND c2<='9' AND c3='0' AND c3<='9') THEN
```

```
              test[i] = (c-'0')*100+(c2-'0')*10+c3-'0';
```

```
            ELSE
```

```
              error("*** TM - encoding error -- test aborted\n");
```

```
          ENDIF
```

```
        ELSE
```

```
          test[i] = c;
```

```
      ENDELSE
```

```
    ENDIF
```

```
    c = getc( fp);
```

```
  ENDWHILE
```

```
  test[0] = i;
```

```
  if (c IS EOF) return(TRUE);
```

```
  return(FALSE);
```

```
END
```

```
/*          ** ascii encoding routine **          */
```

```
/*          */
```

```
/* this routine accepts a number and return its three letters ascii */
```

```
/* representation.          */
```

```
PROCEDURE  ascii_encode(num, ptr)
```

```
INTEGER    num;
```

```

STRING      ptr;

BEGIN
    INTEGER i, pwr;

    IF ( num IS '0') THEN BEGIN
        FOR (i=0; i<3; i=i+1)
            ptr[i] = '0';
        return;
    ENDIF

    pwr = 100;
    FOR (i=0; i<3; i=i+1) BEGIN
        ptr[i] = num/pwr;
        num = num - ptr[i]*pwr;
        ptr[i] = ptr[i] + '0';
        pwr = pwr/10;
    ENDFOR

END

```

```

/* exception handling routine */

```

```

PROCEDURE error(string_ptr)

    STRING      string_ptr;

    BEGIN
        printf("%s", string_ptr);
        kill(0,9);
        exit(9);
    END

```

```
/* ----- LUT_driver.c ----- */
```

```
#include <stdio.h>
#include "TTX_declare.h"
#include "C_extensions.h"
```

```
/* --- TTX'S GLOBAL VARIABLES --- */
```

```
CHAR_POINTER UP_q_ptr, DOWN_q_ptr, IN_q_ptr;
/* interaction queues for layer under test */
```

```
/* This main program implements the basic logic of the TTX_process: */
/* - picks up input interactions in the TTX_pipe */
/* - activates internal processing of the protocol pseudo processes */
/* - picks up and send output interactions to their destination pipes */
```

```
main( argc, argv)
```

```
    INTEGER      argc;
    CHAR_POINTER  argv[];
```

```
    BEGIN
```

```
        CHAR_POINTER  get_free();
        INTEGER      A_fd, TTX_fd, N_fd;
```

```
        UP_q_ptr = /* queue pointers initialization */
        DOWN_q_ptr =
        IN_q_ptr = NULL;
```

```
        /* configure pipes properly for the T_layer under test: */
```

```
        A_fd = argv[1][1]; close( argv[1][0]);
        TTX_fd = argv[1][2]; close( argv[1][3]);
        N_fd = argv[1][5]; /*close( argv[1][4]);*/
```

```
        LOOP BEGIN
```

```
            IN_q_ptr = get_free();
            IF (read(TTX_fd, IN_q_ptr, INTERACTION_SIZE) ISNOT INTERACTION_SIZE)
                error("*** LUT - TTX pipe read error\n");
```

```
            WHILE (IN_q_ptr ISNOT NULL OR
                   UP_q_ptr ISNOT NULL OR
                   DOWN_q_ptr ISNOT NULL ) BEGIN
```

```
                IF (IN_q_ptr ISNOT NULL) THEN
                    T_PI(); /* Activate transport level processing */
                IF (UP_q_ptr ISNOT NULL) THEN BEGIN
                    /* send out output session level interaction */
                    IF (write( A_fd, UP_q_ptr, INTERACTION_SIZE) ISNOT
                        INTERACTION_SIZE)
```

```

        THEN error("*** LUT - A pipe write error\n");
        put_free(UP_q_ptr);
        UP_q_ptr = NULL;
    ENDIF

    IF (DOWN_q_ptr ISNOT NULL) THEN BEGIN
        /* send out output network interaction */
        IF (write(NLfd, DOWN_q_ptr, INTERACTION_SIZE) ISNOT
            INTERACTION_SIZE)
            THEN error("*** LUT - N pipe write error\n");
        put_free(DOWN_q_ptr);
        DOWN_q_ptr = NULL;
    ENDIF

    ENDWHILE
ENDLOOP

END

#define FREE 0
#define POOL_SIZE 3

static CHARACTER interaction[POOL_SIZE][INTERACTION_SIZE];
static INTEGER buff_pool_status[POOL_SIZE] = {FREE, FREE, FREE};

/* This procedure manages the interaction buffer pool of the TTX_process, */
/* and assigns empty buffers */
/*

FUNCTION CHAR_POINTER get_free()

BEGIN

    INTEGER i;

    FOR (i = 0; i < POOL_SIZE; i = i+1)
        IF (buff_pool_status[i] IS FREE) THEN BEGIN
            buff_pool_status[i] = NOT(FREE);
            return(&interaction[i][0]);
        ENDIF
    error("*** FATAL ERROR -- no free buffers\n");
ENDIF

/* This procedure returns empty interaction buffers of the TTX_process */
/* to the buffer pool */
/*

PROCEDURE put_free(buff_ptr)

CHAR_POINTER buff_ptr;

```

```
BEGIN
```

```
    INTEGER    i;
```

```
    i = (buff_ptr - &interaction[0][0])/INTERACTION_SIZE;
```

```
    IF      (i < POOL_SIZE && i >= 0) THEN
```

```
        buff_pool_status[i] = FREE;
```

```
    ELSE    error("*** FATAL ERROR -- invalid pointer\n");
```

```
END
```

```
/* This procedure is responsible for managing the local interaction */
/* processing of the T_layer pseudo-process                               */
```

```
PROCEDURE T_PI() /* transport process interaction routine */
```

```
BEGIN
```

```
    INTEGER    result;
```

```
    STRING     buff;
```

```
    buff = get_free(); /* free buffer */
```

```
    result = T_service( IN_q_ptr, buff); /* call to the Transport    */
                                          /* layer protocol module's */
                                          /* interface procedure      */
```

```
    CASE (result) OF
```

```
        /* send output to the appropriate queues */
```

```
    case UP:
```

```
        put_free( buff);
```

```
        UP_q_ptr = IN_q_ptr;
```

```
        IN_q_ptr = NULL;
```

```
        EXIT;
```

```
    case DOWN:
```

```
        put_free( buff);
```

```
        DOWN_q_ptr = IN_q_ptr;
```

```
        IN_q_ptr = NULL;
```

```
        EXIT;
```

```
    case BOTH:
```

```
        UP_q_ptr  = IN_q_ptr;
```

```
        DOWN_q_ptr = buff;
```

```
        EXIT;
```

```
    OTHERWISE:
```

```
        put_free( IN_q_ptr);
```

```
        put_free( buff);
```

```
        IN_q_ptr = NULL;
```

```
        EXIT;  
    ENDCASE  
END
```

```
/* Exception-handling routine */
```

```
PROCEDURE  error(string_ptr)
```

```
    STRING  string_ptr;
```

```
    BEGIN
```

```
        printf("%s", string_ptr);
```

```
        kill(0,9);
```

```
        exit(9);
```

```
    END
```

END

11-06-86

FIN