NOTICE

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

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

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

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

AVIS

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

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

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

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

Protocol Engineering Issues for Open Systems Communications

By

Syed A. Aleem

A thesis submitted to
the School of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science
August, 1990

Department of Computer Science
University of Ottawa
Ottawa, Ontario
CANADA

ISBN 0-315-68037-7

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

I hereby declare that I am the sole author of this thesis. I authorize the University of Ottawa to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Syed A. Aleem

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

Syed A. Aleem

Abstract

The complexity of end to end communications services offered by communications protocols has increased tremendously in the past decade. With the advent of *outband call control protocols*, *high speed networks* and *cellular communications*, the communications service providers are likely to offer highly imaginative services in the near future. As a result global communication will undergo a metamorphism and the social dependence on services may re-configure human activities.

The author has been personally involved in the protocol engineering process from the specification to delivery of an end product to the customers. The protocol engineering process is complex and expensive. There has been a strong trend in the research world to encourage the automation of protocol product development using formal description techniques and tools. However, much of this work is less concerned with pragmatic issues and real protocols. In order for formal description techniques to be practical, real design criteria must be taken into account.

In this thesis we present real design criteria and design building blocks in a highly procedural way starting from the specification to the implementation stage.

A subset of a real protocol, namely ISDN Q.931 [CCIT3], written in informal English will be translated to a formal specification using Estelle as the formal description technique. A means of incorporating the design criteria into the formal description of the protocol is presented and illustrated with Q.931 as example.

This thesis will also derive the services a protocol expects from the operating system. Those services will be summarized and standardized in order to limit variability in the design process of a protocol. The system interface which has been a major source of variation in protocol implementation, will reduce to a set of a few well defined standard system calls by assuming a proposed standard operating system interface. This interface standard appears to be sufficient to allow the automatic development of complete protocol software.

We believe that the entire process of understanding, designing and implementing real world protocols will become more amenable to protocol engineers based on this thesis and subsequent extensions to some parts of this approach.

Acknowledgements

I would like to express my immense sense of gratitude to my thesis Supervisor, Rev. Dr.Robert L. Probert, for his guidance, encouragement and advice throughout the course of my study and research. In spite of his busy schedule, he was approachable and helpful to me both on and off the campus, as I was employed at Toronto.

I am very thankful to my wife Shoukat Fatima Aleem for her moral support, patience and encouragement, who in spite of our two little sons, was very understanding and co-operative.

I would like to thank IBM Canada Ltd., and my manager Steve I. Fekete for permitting me to return to school on a sabbatical leave of absence.

Dedication

In memory of my father Late Dr.Syed A. Gaffar.

Table Of Contents

Chapter 1

1. Introduction

More and more of today's applications depend upon open computer communications. The variation in behaviors across computers manufactured by diverse organizations has mandated a requirement for systems which are open with respect to communications and interworking. In fact this generation has seen the society change from manual transactions to a sophisticated instantaneous computer information exchanges involving distributed processing and off site computers. Beneath all the modern computer applications there is a complex heterogeneous computer communications network.

In the past two decades, enormous advancements have been made in both computers and in computer communications. Adhering to correct and effective Communications Standards has become key to successful implementation of open systems.

The advancements in communications have enabled the end to end interworking of heterogeneous systems and has resulted in the birth of open systems communications architecture and standard communications protocols. Examples of successful architecture are Open System Interconnection Reference Model and more recently the ISDN standards (treated as a new architecture by a few).

The past decade has seen the evolution of numerous communications standards.Manufacturers, network providers, and customers all incur significant costs in developing, implementing, testing and maintaining communications standards and therefore have a vested interest in promoting productivity in the protocol engineering process.

A serious problem in improving the protocol engineering process is representing design decisions in functions within the protocol specification. This thesis will attempt to isolate such design issues and provide a design phase a mechanism to support automation of the protocol engineering process.

1.1 Motivation and Overview of the Protocol Engineering Process

The fundamental concepts of design of a communication protocol in a real environment are not adequately developed and discussed in the literature. The entire protocol engineering process is not well documented or understood. There is a definite need for automating various facets of the protocol development process for economic reasons and to enhance the timeliness of delivery of new protocols and services.

A protocol development process is similar to any conventional software development life cycle. The difference between the Software Development Process Cycle and the Protocol Engineering Process is as shown below:

The major phases of the software development process are :

1. Product Planning
2. Product Design
3. Product implementation
4. Product Testing and
5. Product Maintenance (Enhancements and corrections)

The major phases of a protocol engineering process are:

1. Service Definition
2. Protocol Design - includes Validation
3. Protocol implementation and
5. Protocol Testing - includes Conformance Testing
6. Protocol Maintenance (Enhancements)

Note that there are significant differences between the software development process and the protocol engineering process. Protocol engineering consists of certain sub-phases or activities which are specific to protocols. For example, the specification and design phases contain an additional Validation Phase while the Testing phase contains an additional Conformance Testing Phase.

A wave of interest in automating protocol development is occurring in both research and development [Nash83, BoGe87, TaSh88, SiCh89, SiBl90]. The development of formal description techniques and well-defined theories for specification, design, validation and testing of protocols is assisting the automation process. One motivation for our work is to combine expertise in protocol development, and the effective use of formal description

techniques and methods. As well, we hope to make the application of our method clear by applying it to a real protocol of widespread interest.

1.2 Contributions of the Thesis

The Major Contributions of the Thesis are:

1. A systematic methodology for *Analysis and Design* in the protocol engineering process and its presentation in the form of comprehensive steps.
2. Methodology and supporting standards for *minimum system services* and for a *standard system interface.*
3. Means of automating the methodology leading to automatic generation of complete code for the protocol.
4. Support of feasibility of methodology by presenting a detailed application to a real protocol of widespread interest, ISDN Q.931.

The Minor Contributions of the Thesis are:

1. Comparative study of formal and informal specification techniques from a pragmatic point of view.
2. Identification of the main issues in designing a parser (encoder/decoder) for the protocol.
3. Identification of issues in translating a real protocol using Estelle.

The following paragraphs will elaborate on the above contributions.

More specifically, in the research carried out for this thesis, a survey of the protocol design and implementation literature was performed. The literature on design of service specifications contains the high level concept of protocol inter-layer service design. However, the feasibility of the suggested design techniques are not supported with examples. In this thesis we apply protocol design and service specification design techniques from the literature to real world protocols. In addition we have augmented the design process to include local environment dependence on design.

The basic approach of this thesis is to relate considerations from the execution environment and protocol specification to the protocol service requirements in a highly procedural manner. The rules laid out in this thesis are fundamental to a logical and efficient protocol implementation process.

Specifically the thesis will promote the use of a formal description technique, Estelle, for protocol engineering. In order for complete protocol code to be generated automatically, the data structures which represent the static and dynamic behavior of a protocol must be completely and formally defined. Use of Estelle promotes the precise representation of static and dynamic structures of a protocol. Once an implementation based formal specification is obtained the production of automated protocol code is feasible.

In order to provide a realistic assessment of our approach, ISDN Q.931 will be used as an example. We will specify a simplified ISDN Q.931 (enough to establish a voice or data call) protocol in Estelle. The derivation of such an FDT in itself is very complex. Thus, the representation of the simplified Q.931 protocol in Estelle and the analysis of the features of Estelle which are used is also a significant contribution of this thesis.

1.3 Organization of the Thesis

Since a formal specification approach is employed in this thesis we first evaluate the suitability of a formal description technique over an ad hoc technique both from the specification and the implementation points of view. Thus the issues related to specification of a protocol, the properties of protocols which should be specified, the merits and demerits of formal and informal specifications are discussed in Chapter II.

In Chapter III we characterize the design process of a protocol. The major issues which should be considered before a protocol is designed are systematically presented. A comprehensive analysis of issues related to parser design, service specifications and operating system interactions is presented. An in depth analysis of the Service Specifications is carried out and used to generate the Service Specifications of a LAPB protocol as an example.

In protocol automation literature, the system interface is often ignored. Chapter IV presents a survey of the literature on the protocol automation process. The services which a protocol expects from the operating system are identified to help create a standard system interface.

Chapter V is devoted to introducing to ISDN in order to introduce the reader to the relevant concepts of ISDN and Q.931.

In Chapter VI, we walk through the design steps in our approach as applied to Q.931, namely analysis of protocol documentation, definition of peer to peer state machine etc..

In Chapter VII, an introduction to important aspects of Estelle is provided. Analysis of Estelle's features useful for specifying ISDN Q.931 is also provided and in the process features which are desired but are not available in Estelle are highlighted.

Chapter 2

2. Protocol Specification Issues

Protocol specification is one of the most difficult and ambiguous aspects of a protocol engineering process. The author views the difficulty in specification, is not due to the nature of the protocol but due to shortcomings in the available techniques. In general for any two systems to communicate with each other a local protocol can be architected and specified before it is implemented. On the other hand for a system to communicate with any other system in the Communications Open Systems, it is required to follow certain protocol standards which are laid out by the international bodies such as CCITT, ISO, etc.

Considering the critical nature of present day applications such as electronic banking which depend upon underlying protocol reliability and integrity, the task of precisely interpreting the standards demands extensive experience and teamwork among experts.

2.1 Formal Specification Techniques

It is very difficult to formally specify a modern day protocol such as ISDN which is very complex. The process of formal specification of a protocol using Formal Description Techniques has evolved over a period of time [Piatk86], [BoLo90]. The following formal models are widely used in formalizing the protocol specification:

1. State Transition Models
2. Programming Language Models
3. Hybrid Models

2.1.1 State Transition Models

These models are simple and easy to visualize. It makes use of the fact that it is easy to capture the behavior of a protocol description using a state transition system. A protocol entity can be described in terms of a set of transitions of a suitable finite state automaton. An element of such a set describes an input and or / output communications event. The advantage of such a model is that, it is simple and many general properties of a protocol can be verified. This technique is however not suitable for describing large and complex protocols. Specification techniques such as Communicating Finite State Machines (CFSM) [Boch78] and Petri Nets [Diaz82] make use of this model. Petri Nets are mainly used as a model for performing certain analysis on protocols rather than as a specification language.The communicating finite state machines are shown to be very useful for specification [Boch78] [SiKa82], analysis[Boch78], [BrZa83], [ChoGo84], [Zafi80], and synthesis [BoSu80], [GoYu84], [ChoGo84].

2.1.2 Programming Language Models

A communication protocol is programmed in software on a system using certain standard algorithms and procedures. Therefore, a communication protocol can be represented using any high level programming language. Specification techniques such as Hoare's communicating Sequential Processes (CSP) [Hoar85] , and Milner's Calculus of Communicating Systems (CCS) [Miln80] use this technique. ADA and Prolog have also been used to specify protocols. Even though ADA has a potential to become a protocol specification language [CaDu86] [YeGe82], it is not very popular, perhaps because it was designed by members of non communication environment.

2.1.3 Hybrid Models

These models combine the above two techniques. In this model, the state transition models are assigned variables and included in the algorithmic representation of the communicating system. The states of this model represent the dynamic behavior of the communication protocol while the variables are used to represent the attributes required for the program like aspects of the protocol. Specification techniques such as IBM's FAPL, DoD protocol specification technique, ISO's specification techniques such as Extended State Transition Language (Estelle) [ISO3] and LOTOS [ISO8], CCITT's specification technique such as Specification and Description Language (SDL) [CCIT4], [DiPi83] belong to this category. There are other languages such as IC*[CaCo88] designed to create an environment for the implementation lifecycle of communications protocols, parallel machines and distributed systems.

2.2 Literature on Specification Languages

FAPL was successfully used for the semi-automated proto-typing of the Systems Network Architecture at the IBM [IBM80] [PoSm82] [Nash83]. The specification of SNA in FAPL was compiled to produce high level languages such as PL/I and PL/S [IBM74, IBM79], which were then compiled using the existing compilers for PL/I and PL/S.

There are many other specification languages which are either local to an organizations or were created by researchers such as, PDIL[Ansa82] [Ansa83], SL1[ExPo82] and LC/1[AyCo82].

Estelle, LOTOS and SDL have been accepted as international formal description techniques to specify communication protocols. Even after the acceptance of the above languages as standard formal protocol description languages by the standard bodies, their complete potential is not yet realized by the protocol development community due to many reasons, one of the simplest reason being the reluctance on the part of the developers to learn a new technique and develop tools for it.

Each formal description technique has its own advantages and disadvantages over the other. As a result protocols are now specified mainly using English verbose description supplemented with SDL diagrams which depict the State Transition part of the protocol.

Since the protocol standards are described in English, the representation is not accurate. There are many inter dependencies within the protocol which are difficult to describe with clarity. If the standards groups succeed in describing the protocol in English, it will be with too many cross references. Therefore the understanding of the protocol specification is very complex, time consuming, error prone and expensive as compared with other aspects of protocol engineering process.

Specification languages for other facets of protocol engineering are also available. The requirements for a Test Specification Language for Protocol Implementation Testing is given in [PrUr83].

A useful survey of Issues and Experiences related to Formal Specifications of protocols is given in [BoLo90].

The following section presents a detailed summary of issues pertaining to usage of Formal Description Techniques.

2.3 Analysis of Issues Pertaining to FDTs

Specification of a communication protocol is a complex activity. The understanding of certain properties which are considered desirable within an operational protocol will help in visualizing the complexity of a formal protocol specification. It is not

the intention of this thesis to elaborate on the above properties, however a brief definition of the protocol properties is provided in section 2.4 and it is recommended to make use of available literature in the field, such as - algorithmic procedure for checking the logical correctness of communication protocols [UyLa90], a comparison of last 10 years for protocol verification and a method to develop and implement a logically sound and verified communications protocols [Miller90]. In order to specify a protocol such that the desirable properties of the protocol are embedded within the specification and in order for the specification to be useful in checking for the desirable properties of the protocol, the specification language itself should meet certain requirements [DiPi83, Boch90] given in section 2.3.1. A very unique analysis of fundamental attributes required to characterize and compare formal protocol specification techniques is given in [VePi86].

2.3.1 Requirements to be satisfied by a FDT

The CCITT Special Rapporteur's Group for Question 39/VII has defined a set of requirements for the specification language to guide the work on formal description techniques and to assist in the evaluation of candidate languages. The requirements are the following :

Applicability

The formal techniques used for specifying the language should be applicable to all layers of the Reference Model. The formal description language should contain the OSI concept of interaction between various components of the Reference Model. The specification should specify the structure of the component which makes up the system being described.The specification language should be able to specify the allowed interactions and their data parameters and the control flow governing those interactions.

Readability

The degree of abstraction, modularity and presentation formats should be acceptable for easy human comprehension. The specification should serve as a good mental model for implementation. The right degree of high and low level abstraction should be exercised to maintain modularity in the specification. The language should have techniques to contain complexity. One easy way to handle complex specification is to modularize it with comprehensive reasoning. It may be necessary to adopt dual specification scheme, to suit human and machine understanding.

Formality

The language should be precisely defined and should provide accurate, complete, consistent and correct specifications.It should be able to handle the definition of all the actions, sequence rules, data and control communication procedures.

The formalisms should serve to determine the completeness and accuracy of service and protocol designs, and for testing the conformance of implementations. The formalism should be concrete for performing automated syntax checking, consistency checking, and to permit automated protocol verification and implementation.

2.4 Desirable Properties of a Protocol

The desirable properties of a protocol fall within the umbrella of protocol verification.Verifying a communication protocol means ensuring that the protocol is free of logical errors prior to implementing it. The verification of a protocol specified using a programming language is difficult to automate because of the insight required [HaOw83, SaSc84]. Desirable properties of a protocol which should be verified are classified into two categories: *general* and *specific* properties and are discussed in [GoRo84], [RuWe82], [WeZa78]. A general definition of the above properties and Protocol Synthesis and Analysis is given in section 2.4.1 and 2.4.3 respectively.

2.4.1 Desirable General Properties of a Protocol

General properties are desirable in all protocols and are an integral part of every service specification [Zafi83]. Some examples of the General Properties of a protocol are absence of deadlocks and of unspecified message receptions. General properties are equivalent to the syntactic correctness of a program. The desirable general properties of a protocol are:

Freedom from Deadlock

A system will never get into a state where no more transitions can occur . If such a global state exists then the protocol will stay in that state indefinitely. Such a state could be as a result of an error in the specification.

Freedom from Unspecified Message Reception

It is an error if the protocol receives a message in a specific state and does not know what to do with it. It is because of a design error in the protocol.The protocol behavior may become unpredictable after such a message is received.

Freedom from Livelocks

A Protocol is said to allow livelock if it can reach a state that can be subsequently forced to repeat the same sequence of transitions forever.

Freedom from Non-Executable Sections

A protocol is said to contain Non-Executable Sections if it contains certain global states and transitions which can not be reached and fired respectively by either normal or exceptional messages. The code which contains such transitions is called the dead code.

Completeness

It means that a protocol can accept all possible inputs in each system state.

Termination

A protocol will reach the terminating or final state when started from its initial state.

Cyclic Behaviour

A protocol can repeatedly progress from one state to another state, if so desired.

Boundedness

The total number of messages in channels between layers at any particular time will never be unbounded.

2.4.2 Protocol-Specific Properties

Protocol-specific properties are the services expected from the protocol by the designer of the protocol. For example, an ISDN Q.931 should negotiate and seek a B channel. Another example is that a data transfer protocol on an ISDN B channel should transfer data correctly.

Protocol-specific properties are similar to semantic properties of a protocol. Specific properties are usually broken down into *Safety Properties* and *Liveness Properties* [Hail83].

Safety Properties

The Safety Properties in general state that nothing unexpected or bad will happen in the life of the protocol for an established end to end call. Safety properties are similar to the partial correctness used in the program verification area[Pehr89].

Liveness Properties [Pehr89]

This property is indicative of the service provided by the protocol over a period of time, that is the protocol will make progress. For example, for an ISDN Q.931 call, once the B channel data transfer has commenced, and assuming the safety properties are satisfied, then data messages are exchanged correctly between the peer protocols provided the protocol makes progress.

Liveness properties express future behavior of the protocols and they can be stated using temporal logic [Pneul77] which enables one to reason about the future states of the protocols.

2.4.3 Protocol Synthesis and Analysis

For a formal description technique based specification, an automated validation process is usually carried out at an advanced state of specification development, while protocol *synthesis* is performed at an early stage of specification development. A comprehensible survey and assessment of Synthesis of communications protocols is done in [SaPr89].

Protocol Synthesis:

It is a process where in a protocol is constructed from its usually informal specification by the application of certain design rules which govern the protocol. The so constructed protocols should have the above *desirable properties* [BrZa80, Zafi80, GoYu84]. An automated synthesis based on a set of production rules is given in [Zafi80].

Protocol Analysis:

When a specification is already provided, it is possible to prove it satisfies certain desirable properties [Zafi83] by performing an analysis on the specification of the protocol. The two popular methods are state space exploration [West86] and program proving [SaSc82].

2.5 Specification of Conventional Implementations

By now numerous internationally standardized and proprietary protocols have emerged such as Start_and_Stop, Bisync, SDLC, X.21, X.25, LAPB, LAPD, Q.931, SNA, Ethernet, LAN Protocols (Slotted Ring, Token Ring, Token bus and CSMA/CD) and many protocols pertaining to higher layers of ISO open systems architecture. They either coexist with contemporary protocols or have been discarded because of the emergence of the protocols which provide more modern services.

Most of the above protocols were implemented by numerous organizations and a majority of them were implemented by making use of the conventional English specifications. After the experience gained by the implementors, going back to most of those protocols will seem easier- it reflects on the specification techniques which were employed. Because of the ambiguities of the specification the entire protocol engineering process was often difficult and expensive.

The following section will summarize the difficulties encountered in the protocol engineering process as a result of implementations of informally specified protocols. The author has surveyed the literature [BaCo90, FoSa90] and has actually worked on protocol design, development and testing which has enabled him to arrive at the following observations.

2.5.1 Drawbacks of Informal Specifications

1. Informal specifications after all depend upon the clarity achievable by humans, unlike formal specifications which are governed by set of rules. But there is no yard stick to measure the clarity and there are no sets of rules to adhere to, therefore the interpretation of informal specifications will vary considerably. The understanding of the semantics of the protocol can be accomplished only after painstaking cross references and binding of ideas.
2. The implementation tends to have a good part of behavior conform to specification but it is almost always certain to contain some undesirable errors and behaviours leading to unpredictable service due to errors and omissions in informal specifications.
3. Informal specifications mainly describe only peer-to-peer communications. Even though a set of well defined inter-layer primitive interaction rules are laid out by ISO, the actual inter-layer interactions are left open for implementors choice. It is this area which is often complex and a failure here will affect the peer-to-peer communications.

4. There are many desirable properties of protocols (as defined in section 2.4) which cannot be verified as the informal specification cannot be directly tested.
5. Even though protocols developed from informal specification do exist today and are often highly reliable, this does not show the number of iterations and updates it has undergone before it has come to the present stage. Every protocol developed from informal specifications will experience interactions between the standards groups and implementors in order to sort out ambiguities. These interactions are expensive because of the delay involved. In general, the conventional method is error prone and economically not justifiable.
6. Any late modifications to the informal specification will be very expensive.

2.5.2 Merits of Formal Specifications

1. Since a well defined set of rules and conventions are followed in the use of a FDT, the specification will be more consistent and less ambiguous. The presence of rules for a formal specification makes it highly procedural with chronological stepwise refinements.
2. The purpose of specification is to provide a global consistency in implementation. Since an FDT is procedural, it is much easier for a developer to picture it and for the computer to execute it.
3. Logically structured functional and procedural rules are often close to an automatic implementation. Therefore FDTs have been successfully used to produce a good proportion of executable protocol code.
4. The participation of members of various phases of protocol engineering process within an organization will assist in the interpretation of the formal specification in an identical way, which eliminates ambiguity, inter-dependency and is fast and economical.
5. Effect of late modifications will be comparatively smaller if semiautomatic implementation methods are employed.
6. With the availability of compilers for formal specifications, the critical information pertaining to protocols such as transitions, events, states, timers etc., are captured in well defined data structure. It is therefore much easier to write other tools and attain automation for protocol engineering process such as validation of specification, test case generation etc.,
7. Certain constructs are available with FDTs such as the concept of communications via certain interface points, the parallel processing of certain functions which are

independent of each other, structuring of a large specification into smaller functional parts etc., which will enable a specification to represent the behavior concisely.

8. An FDT is able to easily accept certain extensions within a local environment such as Estelle* [Cour87] and SDL*[Diet90].
9. A survey of literature on use of extended versions of Estelle- Estelle*[Cour87] reveals that a number of research and development projects have successfully used Estelle* which has enabled them to specify their distributed system accurately in their local environment. Estelle's power as a technique for the specification of general distributed systems is recognized [Linn88] [Budk87].
10. It is easier to simulate a protocol using FDTs before it is implemented for the logical correctness [JaBo83] [UrPr84] and performance analysis of the final protocol.

Additional benefits of formal description techniques are given in [BoBr87] [ISO8] [ISO9].

2.5.3 Demerits of Formal Specifications

1. Formal specifications may tend to over specify . A FDT such as Estelle tends to incorporate details which are normally implementational issues. This can be constructed as both a positive or a negative aspect. The positive aspect is that, ISO lays down a standard architecture and a standard form of communication services between layers, it is to the advantage of an implementor if he can find a specifications which is closer to implementation.
2. The formal specifications could become very large.
3. The semantics of a formal specification are involved, and the language used for formal specification tends to be quite complex.
4. Writing good compilers which accept all the syntactic and semantic behavior of a formal specification languages is quite complex.
5. Many problems within the usability and functional ability of the FDT may be exposed only when it is used by a large number of organizations for many projects. The problems encountered normally require a non formalized local solution because of the delay involved.
6. There is a one time effort involved in developing dependable tools for FDTs.
7. The protocol engineers should be trained for effective usage of FDTs.

2.6 OSI Architecture

The protocol specification design and its implementation is directly influenced by the protocol architecture. The protocol design and implementation approach in this thesis is based on the OSI architecture [ISO1, ISO2, Zimm80].

The following terms are widely used in OSI terminology:

System

A set of one or more computers and associated software, peripheral devices, terminals and terminal operators etc.

Reference Model

The functional grouping of a protocol into a model which consists of multiple layers.

Service

Refers to the abstract capability provided by one layer of the OSI reference model to its upper layer.

Protocol Conformance to OSI Architecture

Means that the functional behavior of a developed protocol within a layer with the remote peer layer will be in a standard way. In other words, irrespective of the method of implementation, the function of any particular layer should remain a standard.

OSI has provided a set of ***recommendations*** to facilitate the interconnections of computer systems. OSI reference model divides the area of standardization for interconnection into a series of layers of manageable size and function.

Each layer is responsible for certain logical functions. There are seven layers within the OSI model as shown in figure 2.1, and the entities within each layer will communicate with the peer entities in the corresponding peer layer, as shown in figure 2.2. The functions covered by the seven OSI layers are in[Datapro89].

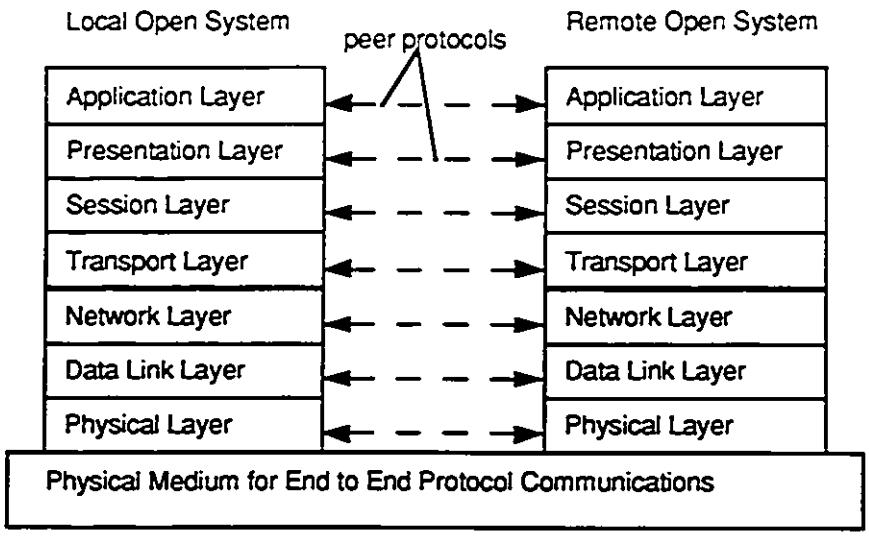


Figure 2.1: OSI Reference Model Architecture

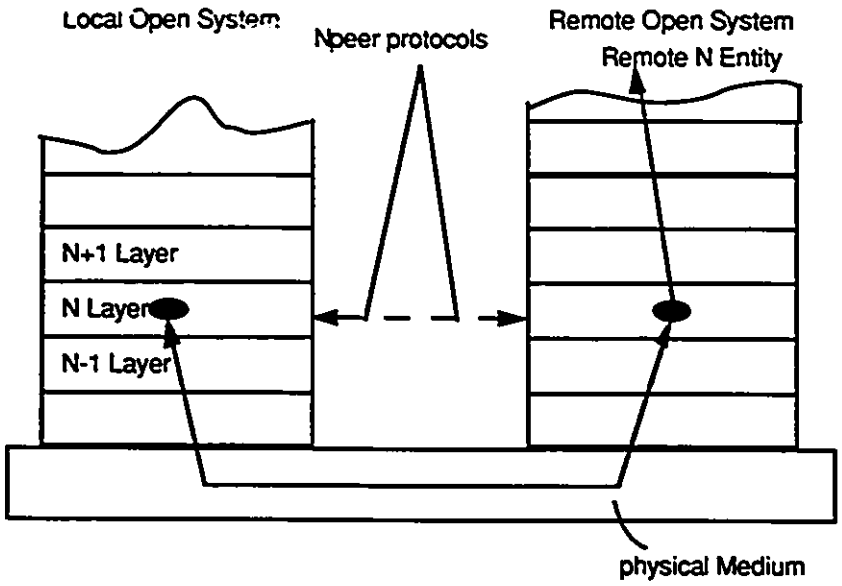


Figure 2.2 Communication path between peer Entities

A protocol need not have all the seven layers of the OSI model. Normally, the protocol starts growing from bottom up. By the time the standards are fully defined for all the layers, it will take a long time and some painstaking global mediation and negotiation.

The OSI recommendations define the services provided by the underlying (N-1) layer to the upper(N) layer. It provides for the designers of N layer a definition of the N-1 Service which exists to support the N layer protocol and for the designers of the N-1 protocol a definition of the services to be provided. The N-1 layer is called the ***provider*** of the Service and the N layer the ***user*** of the underlying service.

The protocol standards define only the peer-to-peer interactions in an OSI architecture. It can be seen in the protocol design phase in chapter three that, there is an inter-layer communication between any two adjacent layers of an OSI architecture.The protocol standards do not specify the inter-layer communications. However, OSI has outlined the general process of inter-layer communications [ISO2], but doesn't specify the parameters, structures or the design process behind the birth of such primitives. The author has noticed, for example in the ISDN CCITT I.441 and I.451 that certain key inter-layer primitives which are responsible for characterizing the peer-to-peer specification are clearly spelled out. These primitives also depict certain major parameters which are communicated between the layers (either specifically or through certain implied names which are indicative of the services they provide). In a real development environment, it might be necessary to add more primitives and parameters to already provided primitives. Almost without exception data structures to support the above primitives will have to be designed. Therefore the design and implementation of the same protocol could vary widely.

Since protocol standards don't specify or only partially specify the inter-layer communications, the specification of inter-layer communications will now become a part of the protocol design and implementation process rather than the peer-to-peer specification process. Chapter 3 will cover the area of design of inter-layer service specifications.

Chapter 3

3. Protocol Design Issues

3.1 Terminology

Protocol Stack

A collection of OSI layers is called a *Protocol Stack*. It is not necessary that all the seven layers of the OSI architecture should be present in a system: the number of layers in the stack are a function of the services required from the underlying layers and the network (as a whole) as viewed by the top-most user (end user) of the stack.

Network Stack

The design requirements of lower layers of a protocol stack may be somewhat different as compared with the higher layers. Therefore the bottom three layers, namely, the network layer, the data link layer and the physical layer are collectively called the *Network Stack*.

Underlying Stack

The term *Underlying Stack* is used always in reference to a layer. If N layer is the referred layer then the term *Underlying Stack* means all the layers below and including the N layer.

Primitives

The inter-layer data units between any two OSI layers are called the *inter-layer primitives*, or simply *primitives* or *service primitives*.

End User Interactions

The interactions of the user with the topmost layer of the underlying stack are called the *end user interactions*, or simply *user interactions*. End user interactions configure the protocol behavior and extract services from the underlying stack. The parameters passed by the end user will directly affect the various protocol layers of the stack. Therefore, the parameters which the user sends should be highly designed from the point of view of the protocols of all the underlying layers. For example, in an ISDN implementation, a specific B channel could be passed as a parameter to the Q.931 layer, or the Q.931 code should compute and decide on a B channel from the available ones. Depending upon the above selection, the design of Q.931 will vary considerably.

Peer-to-peer Data Units

The *peer-to-peer data units* are exchanged between peer entities. In the contemporary technology and therefore in this thesis, the peer-to-peer data units are denoted by any of the following terms: *PDUs*, *Messages*, or *Peer-to-peer data units*.

Service Specifications

For any N layer, the *service specification* defines the inter-layer interaction primitives that are sent to the user entities in the (N+1) layer and the inter-layer interaction primitives that are received from the entities in the N-1 layer.

Protocol Specification

The N layer *protocol specification* defines the behavior of an entity within that layer with the corresponding peer entity in the peer N layer.

Service Access Point

The point of interaction between a service user and the service provider is called the *service access point (SAP)*. When a service user requests a service from the provider, the user layer will provide an identifier to the provider, serving as the SAP Identifier.

3.2 Major Steps in Protocol Design

A number of design methodologies have been suggested in the literature [SeBo86] [Chong86] [BoRa83] [ViLo86] [BoJo79] [DhKo86] [HeRa78] [TeNg90] [ElK90].The

examination of the above literature has assisted in deriving the following steps for the design of a protocol.

The Major steps in the design of any communication protocol are:

1. Careful study of the Specification of the N layer protocol to be implemented.
2. Deduction of high level peer-to-peer state machine tables or diagrams. The tables should show the N layer protocol states, PDUs and the high level action routines.
3. Precise definition of the syntax of PDUs in order to implement the parser (coder and decoder) for the N layer protocol.
4. Incorporation of generic aspects of protocol design due to OSI concepts.
5. Incorporation of generic aspects of protocol design due to Local Design Approach.
6. Design of the inter-layer interactions.
7. Design of Parameters for Service Primitives.
8. Mapping of protocol service requirements into services provided by the operating system for the system under development.

Initially, some of the above steps may be carried out in parallel, as shown in figure 3.1. Once a certain body is imparted into the design process, other protocol factors such as flow control, queuing and de-queing of the peer events and inter-layer primitives, buffer management, etc. should be designed. In the detailed design stage it is necessary to design all the components of protocols, such as system interface, parser(coder and decoder) etc. with all the inputs and outputs. Subsequently, each component of the protocol should be broken down into a number of procedures or functions, and the inputs and outputs of each procedure should be explained and used in the pseudo code.

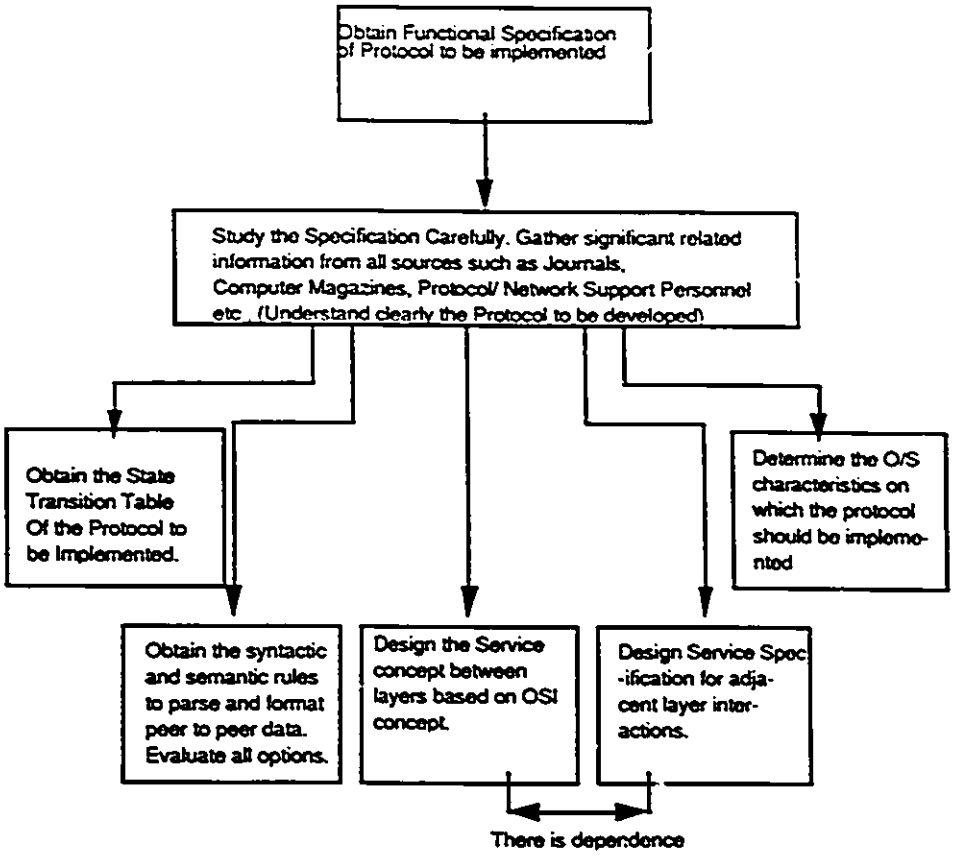


Figure 3.1 High Level Design Strategy For Development Of Protocols

3.3 Study of the Specification

This is the first step of the protocol design and implementation process. The recommendations of the standards groups, which specifies the protocol in English, should be studied carefully and accurately to remove any ambiguity whatsoever before commencing its design and implementation. In addition, a number of researchers have formalized in an FDT many of the functions offered by a protocol. Such formalizations can be used to assist in better understanding of the English specifications. The publications of such research will assist in better understanding of the specifications.

Despite the presence of the private local and wide area networks, the bulk of modern communications is through public wide area networks. It is therefore important to incorporate the local functional specifications of various public voice and data carriers.

The unambiguous understanding of protocol specification is a pre-requisite for successful development of every other step of the protocol engineering process.

3.3.1 Availability of Specifications using FDTs

If the FDT's were very popular, we would have obtained the protocol which is specified in a FDT right from the standards groups, who are responsible for introducing the communication protocol standards. If such were the case, the proposed design methodology can be easily applied to the FDT. But the reality is that not many of the protocols are specified in formal specifications [OSIstat], therefore the developing organizations have to study the informal specification and then translate it into a formal specification. Despite the time and energy spent on translating the informal specification into a formal one, major advantages can still be gained if FDT-based tools for protocol engineering are available.

This phase is critical because the informal specification is translated into a formal one. It should be ensured that all the desirable properties of the protocols which are given in section 2.4 are met by the protocol. In a real communication product development environment, it is very difficult and time consuming to ensure that all the protocol properties of the locally derived formal specifications are satisfied. However, we strongly believe a fairly accurate specification will result in substantial automatic code generation.In this thesis a simplified ISDN Q.931 protocol will be translated from an informal English description into a formal Estelle Specification.

It should be realized that it is very difficult for a pure developmental organization to translate a protocol into a formal specification, as it needs experience and thorough

knowledge of FDT's (which are still evolving). Until such expertise and comfort level is reached it is hard to justify its use from a productivity point of view.

Let us hope that in the future the various formal specifications will become more popular with the protocol engineering community, and the international standards bodies will ensure the availability of a formal specification which satisfies all the properties of the protocols given in section 2.4.

3.4 Design of the Peer-to-peer State Machine

This is one of the most important stages of the design process and a major outcome of step 1 of the study of a protocol specification. Often, only a subset of the protocol may be implemented, therefore decisions must be made on the types of services the corporation wants to provide to its customers by choosing the options available from the protocol specification. For example, if ISDN is developed for a data-only terminal then there is no need for the voice support, and within the ISDN data support there may be no need for network Value Added services. This stage is more than just getting a state machine for states and transitions. The action routines which should be executed on arrival of the peer PDUs, before a transition can occur, will become very complex when all the environmental factors are considered.

To make a decision on peer-to-peer PDUs to be designed (coder and decoder), it is important to know the peer-to-peer state machine which will be implemented. This state machine is also important to make detailed design, such as the availability of worst case scenario resources, such as buffers, timers, etc..

In this thesis the state machine for ISDN Q.931 will be converted into Estelle formalism with a few detailed procedures for major functions and a few Estelle *primitive* procedures, for the action part of the transitions.

3.5 Design of the Parser (Coder & Decoder of PDUs)

It is not within the scope of this thesis to outline the detailed design of a parser. However certain general issues related to a parser are presented in subsequent paragraphs to enable a smooth flow in understanding global protocol design. In section 3.5.1 a means of automating a Parser is suggested for a Network Stack within the Open Systems Interconnection. A useful discussion on parsing of peer PDUs is given in [KrKr87].

The parser is one of the major components of any communications protocol. A Message (PDU) Parser performs syntactic analysis on the received PDU packets in much the same way a lexical analyzer acts on source code in a programming language. Just as a Lexical Analyzer is concerned with the format and not the content of the source code, the

Message Parser should not attempt to interpret the meaning of the received bit pattern. Such semantic analysis is left to the action routines of the transitions (also called *Central Controller Unit* in [KrKr87]). The structure of all the possible messages should be provided by the *Message Format Specification.*

An incoming message will be parsed according to the standards, and tokens are extracted and stored. These tokens are used by the action sub-routines for semantic analysis of the message within its context. If the incoming PDU does not conform to the specified format, an error indication is provided to the action routines. A parser is highly protocol dependant. However, the author has managed to generalize an approach to design of parser by studying the international specifications for X.25, LAPB, LAPD [CCIT2] and Q.931 [CCIT3] protocols.

1. Every protocol consists of peer-to-peer data units such as Messages of ISDN, Packets of X.25, Frames of LAPB, etc..
2. Each of the peer data units contains certain functional partitions, for example, the Information Elements of Q.931 and Commands Fields of LAPB.
3. Each functional group may consist of varying number of octets, for example, single octet or multiple octets of Q.931 Information Elements, one octet address field of LAPB or two octet address field of LAPD.
4. Each octet may consist of functional groups of a single bit or group of bits which represent different aspects of communications, for example, the least two significant bits of the third octet of the Channel Identification Information Element of ISDN Q.931 represents the B channel to be negotiated during establishment of an ISDN call.

Once we have understood the physical structure of peer-to-peer data units and its further physical partitioning down to the bit level within the PDU octets, we should next understand the restrictions imposed by the protocol on the use of above fields.

1. Determine the priority of the direction of parsing of the PDU, for example, find out whether the least significant bit of the last octet should be parsed first or the most significant bit of the first octet should be parsed first.
2. Determine if a standard set of octets (pre-amble) should always be present at the beginning or at the end (post-amble) of the PDU. For example, the control part of X.25 Packets and LAPD frames should come before the data and post-amble. In the event the above pre-amble or the post-amble of the PDU is absent, the PDUs

should be rejected at once instead of parsing the entire PDU completely before it is rejected.

3. Determine the degree of errors permitted. Determine all types of serious errors(non-ignorable errors) such as, absence of Mandatory Information Elements in ISDN, absence of sequence numbers in X.25 Packet Data Unit. If such an error is present, don't parse the remaining PDU. Determine the action to be performed during such an error.
4. Determine all the ignorable errors such as absence of an Optional Information Element in ISDN. During such an error, the parsing of the entire PDU should be carried out. The action sub-routines will take appropriate action on return from parser.
5. Determine the values which every functional bit or group of bits in a PDU can take during interaction with the peer. Transform those values as constants in the protocol code for comparison during parsing.
6. There may be a number of other rules which are specific to the interpretation of a particular protocol PDUs. Add such specific rules to this list.

3.5.1 Automation Of Parser

It is suggested frequently in research literature, more recently in [Boch90b], that protocol specification defines the behavior of a protocol in terms of peer-to-peer interactions through exchange of PDUs. The behavior itself [Boch90b] must be specified in terms of

a. temporal ordering of interactions
b. range of possible interaction parameters
c. rules for interpreting and selecting values of interaction parameters for each instance of communication, and
d. coding of PDUs

It is further suggested that the data structures for decoding the PDUs (b) and formatting of the PDUs (d) for the user applications layer of OSI model can be described using an abstract notation called Abstract Syntax Notation One (ASN.1) [ANS.1]. Other notations which are local to an organization such as AT&T's Connectivity Language LSL [LSL1] are used and they serve the same purpose.

Every protocol specification is incomplete without the unambiguous syntactic and semantic specification of PDUs required for the interactions of the protocol being specified. Therefore, every protocol specification should consist of PDU data types for all the PDUs for the specified protocol.

Because the protocol specification (without PDU data types) can be specified with FDTs such as Estelle, LOTOS, SDL etc., the next step for automation is a standard notation for representing all the types of data a PDU can contain.

The PDUs for the Network stack are normally strings of bits representing alphanumeric characters. But the PDUs for the user layer of the OSI architecture are comparatively complex e.g., IA5 Character set.

ASN.1 is able to represent complex string types of various character types for the user layer applications, and the author believes ASN.1 was standardized to ease the PDU representations for the user layer alone. ASN.1 is always related to the user application layer in the literature.

The author strongly believes that ASN.1 can also be effectively used to represent PDUs belonging to layers of the Network Stack. It is not necessary that all the features of ASN.1 be used.

Other researchers [Boch90b] disclose a number of tools which make use of ASN.1 abstract notation and generate data structures in the implementation language (normally C) and C code for coding and decoding of the peer-to-peer data units. They further suggest a methodology to merge Estelle specifications and ASN.1 specifications to generate automatic protocol code. *However,this code lacks automatically generated operating system interactions.*

The author believes the international standards bodies who design and circulate protocol specifications should also provide ASN.1 PDU specifications along with protocol specifications. Specifications for the network management user layer PDUs which are specified using ASN.1 are available today. The PDUs for the layers belonging to the Network Stack are not specified in ASN.1 by the standards bodies. It is up to the Network Stack designers and implementors to specify informal PDU data types from English specifications into ASN.1, before using tools to automate or semi-automate the code generation.

3.6 Design of a Protocol based on Generic OSI Concepts

3.6.1 Generic OSI Factors in Hierarchic Communications

Since OSI has outlined the requirement, the different layers of a protocol should communicate with their corresponding peer layers: the protocols do so through a set of data units for that layer. A protocol layer may itself consist of more than one entity which will have a corresponding peer-to-peer entity in the peer N layer. For example, ISDN Q.931

may have a separate management entity which could be identified through a unique Connection End Point Identifier.

OSI insists on the layered architecture for the protocol stack. Therefore, peer-to-peer communications are the product of reliable inter-layer services; every layer will communicate with its local serving layer, and this serving layer will communicate with its serving layer until the bottom most layer lacking a serving layer is reached. It is this first layer of the protocol stack, also called the *physical layer*, which will transmit the data units to its peer across the physical media.

When a layer has to transmit the peer data units, it will actually communicate with its server layer, and the server layer in turn will have its own peer data units. Depending upon the type of inter-layer data units received from the user layer and the logical state of the receiving layer, the receiving layer will enclose the received data units in its own header and tail (control information) before delivering it to its server layer. If there are no more server layers then the data units will be sent out on the physical media of transmission.

3.6.2 Generic OSI Inter-layer Primitive Types

A common specification style should be provided. This is done by a unique set of descriptive conventions [ISO2]. The standard introduces the concept of a service primitive as an abstract, implementation-independent element of the interaction between the service user and the service provider [Zimm80]. Four *types* of service primitives are defined in figure 3.2, corresponding to the major interactions between the service users, although every service primitive need not have all the four types.

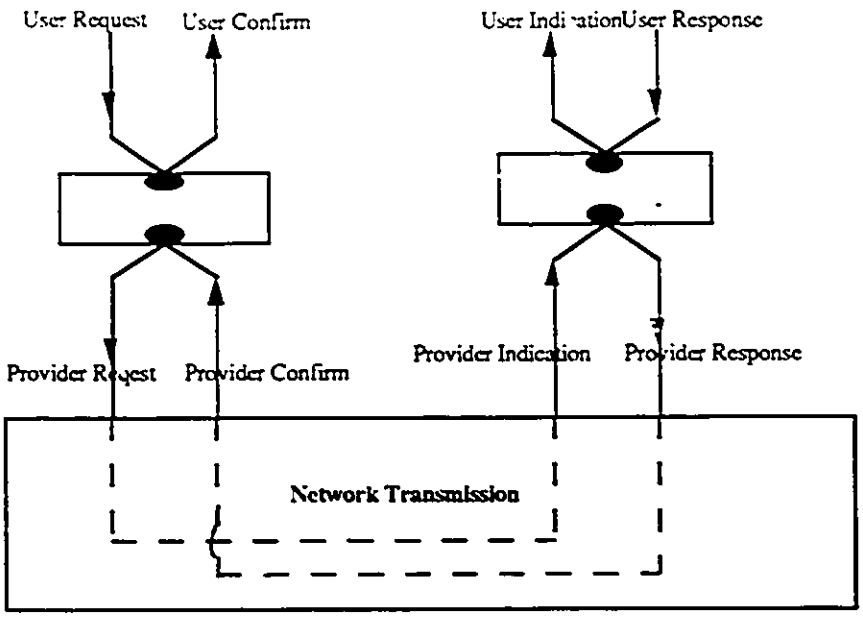


Figure 3.2 Types Of Primitives In A Confirmed Service Specification

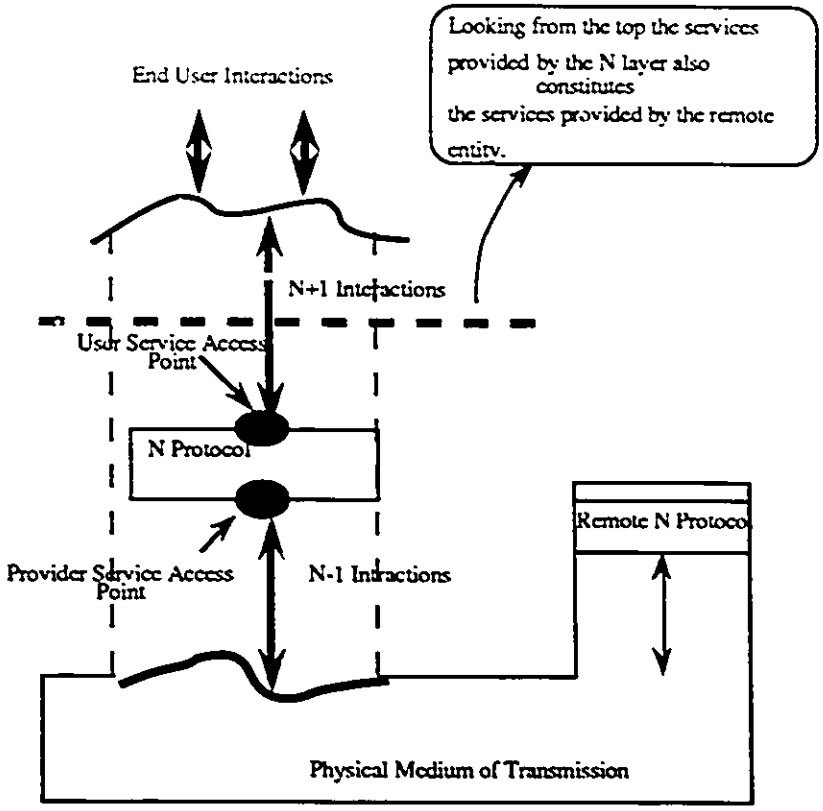


Figure 3.3 The Concept Of Service Access Point Identifier And N Layer Services.

Each primitive is directed from the service user to the service provider or from the service provider to the service user. One or many parameters may be associated with a service primitive. The parameter values associated with the primitive are passed in the direction of the primitive and both the user and the provider can refer to them. The interactions are executed at the common boundary of a service user and the service provider, called the *"Service Access point* "(SAP), shown in figures 3.2 and 3.3.

The available primitive types are:

Request

A primitive issued by the service user to invoke certain functions of the provider layer

Indication

A primitive issued by a service provider either to invoke certain user layer functions or to indicate that the provider layer functions have changed certain global outlook as a result of a peer message or its provider indication.

Response

A primitive issued by a service user to complete the provider function indication at a particular service access point.

Confirm

A primitive issued by a service provider to complete a pending request from the user layer at a particular service access point.

The dialogue between the peer layers can be either confirmed or unconfirmed:

Confirmed

A local service request results in an indication to the peer service user at a particular service access point, which provokes the peer service user to issue a response, which transforms into a confirm at the originating service access point. The types of primitives essential for this type of service specification are shown in figure 3.2.

Unconfirmed

A local service request results in an indication at the peer service user access point and there is no peer user response as a result of this indication.The types of primitives essential for this type of service specifications are shown in figure 3.4.

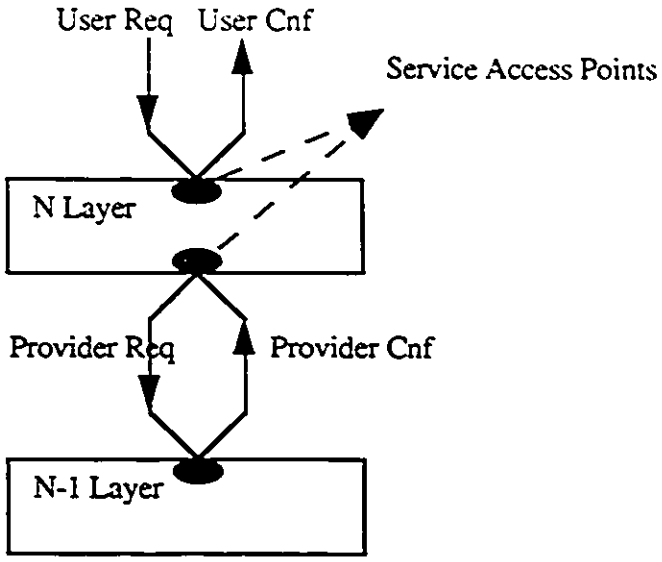


Figure 3.4 Primitive Types In An Unconfirmed Service Specifications

3.7 Design of a Protocol based on a Generic Local Approach

3.7.1 Protocol Layers As Multiple State Machines

CCITT protocol standards comply with the OSI Reference Model. Any protocol is basically state machine driven. In fact, the communications protocols in CCITT are defined as state machines using a notation called SDL [DiPi83]. The state machine, represented by SDL diagrams, is meant to specify communications between entities.

Because adjacent layers communicate between themselves, it is possible to represent the interactions between the layers as a finite state machine. In fact, the design of an inter-layer state machine for a N protocol depends upon the services provided to the upper layer and the services to be expected from the lower layer. Therefore, in order to design the inter-layer state machine, a detailed understanding of the services to be provided to the higher layer, and the services to be expected from the lower layers, is essential.

Any layer of the protocol stack except the top and the bottom layer will provide services to its user layer and extract services from its underlying layer. We can therefore expect any N layer implementation to consist of three state machines. One of the state machines is the peer to peer state machine specified by the standards body and the other two are based on certain service divisions adopted in this thesis. A different service

partitioning based on different local environments and a different provider service will result in different inter layer state machines.

The inter-layer state machine implementation is not mandatory. It is just one of the possible implementations which we have chosen and is discussed in literature. It is also possible to have a single state machine with multiple imbedded transitions in any layer. The imbedded transitions are not very useful for Estelle specifications because Estelle specifications requires transitions which are atomic (that is, once a transition commences there can be no more reception of events within that transition). A sample implementation is necessary for this thesis in order to demonstrate the usefulness of an FDT to accept protocol design in a generic specification.

Therefore, any protocol layer implementation is a three-tuple **[U, S, P]**.

U is a finite state machine which handles Service Startup /Shutdown interactions.

S is a finite state machine which handles establishment of a reliable peer to peer provider service.

P is a finite state machine which handles peer-to-peer communications.

The design of the above state machine transitions, except for the P state machine is now called the design of *inter-layer services* of a protocol, or the *service specifications* of the protocol. The service specification also includes the design of parameters for service primitives.

3.7.2 Communications between Adjacent Layers

It is assumed that each layer is a single task(process) implemented as a single entity, but with multiple state machines, as shown in figure 3.6. Any N layer interactions with adjacent layers will be always through two specific system inter-process communications pipes (channels): one between the U Service Acess Point (SAP) of a N layer with the S Service Access Point of the N+1 layer, and the other between the S SAP of the N layer with the U SAP of the N-1 layer, as shown in figure 3.5. Note that in this figure the inter-layer communications for any N layer seem to enter a N layer protocol through two distinct points but in fact it is through a single entry point, as in figure 3.6. A useful discussion of abstract and implementational protocol queuing concepts is found in [Logrip83].

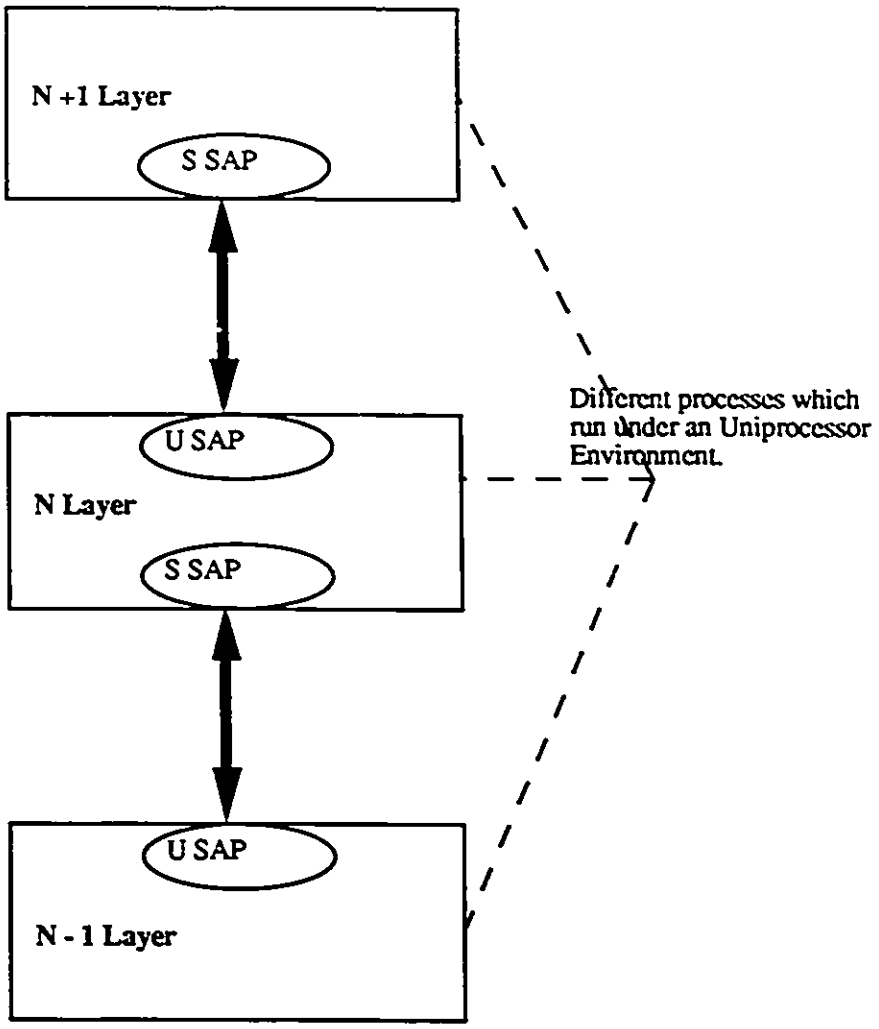


Figure 3.5 Interlayer Communications

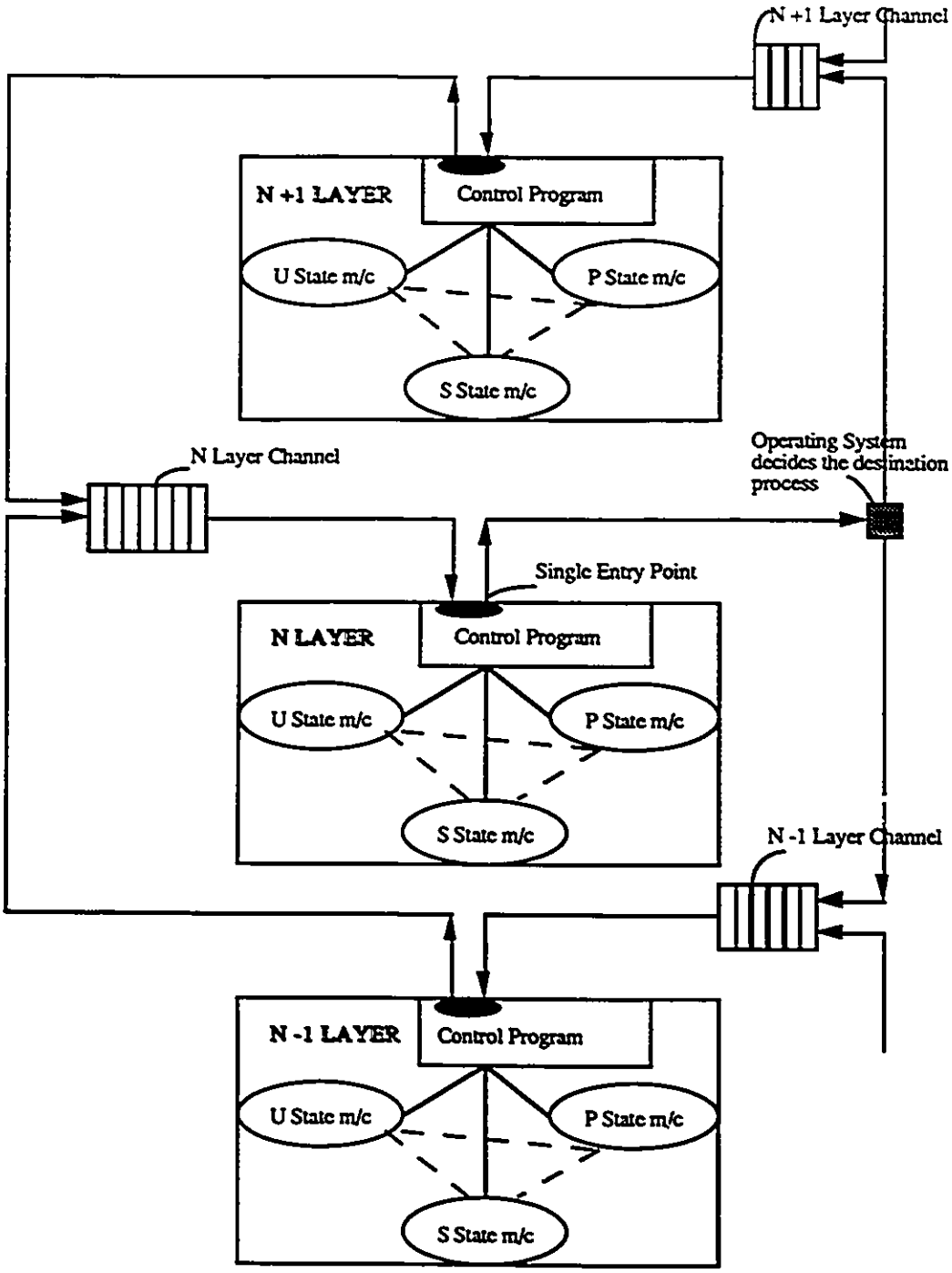


Fig 3.6 Real Interlayer Interactions Through Channels And Single Entry Point

3.7.3 Communications between State Machines in the same layer

In order for an orderly transmission of data between peer entities within the N layer there should be a cohesion between all the state machines local to that layer with the

state machines of its adjacent layers. An error in any one of the state machines will inhibit peer-to-peer data transmission. Sometimes a N layer P state machine may depend on another state machine before performing a valid transition. The inter-state machine interactions within the same layer, shown by dotted lines in figure 3.6, will be through simple procedure calls, while the inter-state machine communication between adjacent layers will be through system calls as suggested in section 3.7.2. Each layer should be developed as a single process with a single entry point, as shown in figure 3.6.

Irrespective of which adjacent layer has sent the primitive, the operating system will always invoke this process at the only entry point. In figure 3.6 a single channel is shown for incoming messages buffered to any N layer. There is a choice of implementation for the operating system to have either one or two channels. The author justifies representing buffers for incoming messages in a single channel for the following reason: whenever the operating system invokes a N protocol layer(process), control is passed to that protocol. Because the system has many tasks, adjacent layers may send messages whenever they obtain, their share of CPU resources(recall that it is a uniprocessor system) and these messages will get queued in a channel(s) for the N process. Messages queued in the channel are ranked in the order of priority. Once the execution of the N protocol exits the main routine, then the operating system will invoke the N layer protocol again and give control to the highest priority message in the channel(s).

When the operating system invokes a N layer protocol it will provide at least one argument which is passed by value to the process. The argument (direct like an array or indirect through a pointer) will point to the main_protocol_control_block which will have information such as the process_id of the sender and the receiver, a pointer to the service primitive structure and a pointer to data (if any) accompanying this primitive, etc..

In the example of 3.7.3.E1, two parameters are passed: one is the process(entity) identifier to which the inter-layer primitive should be passed(its data structure or data type will be present in one of the system or protocol include files included in the Required_Files.h") and the second argument is a pointer to the entity control block.

Example 3.7.3.E1 Main Entry Module to the Process

```
#include"Required_Files.h"        /* Include all system files  */
                                  /*Process and Protocol files */
```

```
                /* main protocol entry requires the process(entity)*/
                /*identifier and a pointer to the Entity Control      */
                /*Block. The Entity Control Block will contain a      */
                /*pointer to the Primitive Control Block and the      */
                /*Data associated with the primitive.                 */
#define SUCCESSFUL  0

main (SYS_PROCESS_ID_TYPE    Entity_Id,
      struct SYS_ProcessCBType   *Entity_CB_Ptr)
{
    int return_code = SUCCESSFUL;
            /* The following procedure ensures that the incoming
            message is indeed for this layer,  ensures the primitive data is
            in accordance with its structure type, and also ensures the
            integrity of received data */

            check_sanity_of_incomming_primitive(Entity_CB_Ptr,
                                                &return_code);

          f (return_code==SUCCESSFUL)
            /* this part acts as the control program shown in figure
            3.6, that is, the state_machine_handler routine will give the
            control of the received primitive information to the
            appropriate state machine*/
               call_state_machine_handler(Entity_CB_Ptr,
                                          &return_code);
                     ...../* more code to be designed */
            /* the following routine may simply be a case statement on
            return code performing the appropriate action, such as
            sending a primitive to the user with the return_code in it*/

        else call_error_handler(Entity_CB_Ptr,
                                return_code);
}
```

The operating system provides the adjacent layer service primitive pointer which is embedded in the entity control block. From the structure to which the above service primitive pointer points, we can write algorithms to check that, the data received is not corrupted or is meant for other processes(caused by programming error). Thus, once we know the structures, it is straight forward to get the Primitive_type, Primitive_Name and the Data carried by the primitive. Whenever there is a service primitive(event) from the adjacent layer, this process will be invoked and, depending upon the incoming event, the process will perform certain actions, change its state if necessary and send out a corresponding primitive if required.

Once the above actions are performed, control will fall out of the main routine. Once this control has lapsed, the next primitive (message) in the communication channel queue will enter the process through the main entry point. The system will determine the source and destination of the primitives and is also responsible for queuing the primitives in their respective processes. If there is no primitive in the queue then the system will not invoke the process therefore the process code will not become executable.

The control program shown in figure 3.6 will determine which state machine will gain control of the incoming message.

The question which arises is how can state machines within the same layer communicate ?. They do so by generating an internal event (internal primitive) within a process. Internal events are exchanged through simple procedure calls. For this, an internal event detector algorithm is needed within a WHILE loop, to ensure that while internal events are generated control should not be lost to the process(main routine). Procedure-oriented communication is within a single process (synchronous procedure calls), therefore the problems associated with asynchronous procedure calls based on "Upcall Mechanism" [Rose90] are absent. The algorithm is shown in Example 3.7.3.E2.

Example 3.7.3.E2 Main Entry Module to the Process

```
#include"Required_Files.h"          /* Include all system files  */
                                    /*Process and Protocol files */
```

```
                /* main protocol entry requires the process(entity)    */
                /*identifier and a pointer to the Entity Control       */
                /*Block. The Entity Control Block will contains        */
                /*pointer to the Primitive Control Block and the   */
                /*Data associated with the primitive.                  */
#define SUCCESSFUL              0
#define PRESENT                 1
#define NOT_PRESENT             0

main (SYS_PROCESS_ID_TYPE    Entity_Id,
      struct SYS_ProcessCBType  *Entity_CB_Ptr)
{
    int return_code = SUCCESSFUL;
    int internal_primitive = NOT_PRESENT;
            /* This procedure ensures that the incoming message is
            indeed for this layer, that the primitive data is in accordance
            with its structure type, and also ensures the integrity of
            received data is O.K.*/

            check_sanity_of_incomming_primitive(Entity_CB_Ptr,
                                                &internal_primitive,
                                                &return_code);


        if (return_code==SUCCESSFUL)
            /* this part acts as the control program shown in figure
            3.6, that is, the state_machine_handler routine will give
            control of the received primitive information to the
            appropriate state machine*/
            {
               call_state_machine_handler(Entity_CB_Ptr,
                                          &return_code);
                     ...../* more code to be designed */
               /* this part of the code sustains the interactions
               between the state machines within this protocol while an
```

```
            internal primitive is produced, the details of which can be
            obtained from the global structure of internal
            primitives.*/

            while ( (internal_primitive == PRESENT) &&
                    (return_code == SUCCESSFUL) )
              {
                    call_state_machine_handler(Entity_CB_Ptr,
                                               &internal_primitive,
                                               &return_code);

              }
        }
        /* the following routine may simply be a case statement on
        return code performing the appropriate action,- such as
        sending a primitive to the user with the return_code in it*/


    else call_error_handler(Entity_CB_Ptr,
                            return_code);
}
```

3.8 Design of Inter-layer Interactions

The peer-to-peer functions of OSI layers should remain a standard, so the manner in which any layer of a particular protocol stack would communicate with its adjacent layers, with the operating system, and with the user interface can be *locally standardized*. The latter services are required only to support peer-to-peer communications. Inter-layer communications are dependent upon generic factors outlined in sections 3.6 and 3.7, as well as the services offered by the underlying protocol layers and the services to be provided to the user layer.

The following naming conventions are used for service primitives in this thesis:

Layer_Identifier	Primitive_Name	Primitive_Type

The Layer_Identifier represents the two letter prefix of the name of the provider layer. For example, the primitives exchanged between the user of the network layer and the network layer itself will be prefixed "NL" (Network Layer); likewise, the primitives between the network layer and the data link layer will be prefixed "DL". The layer identifier will remain the same irrespective of the direction of the flow of primitives.

The primitive name indicates the functions it performs. For example, the network layer will ask the data link layer to disconnect its link to the peer data link layer by sending the "DISC" primitive.

The primitive_type represents the type of the primitive. For example the above DISC request is of the type REQUEST. Extension REQ and REQUEST mean the same type, likewise for other types of primitives.

The entire primitive name, assembled using the above naming conventions, will be NL_DISC_REQ.

The names of the primitives in the diagram or appendices may be shorter than those in the textual explanation. This is done to save space. For example, NL_ACTIVATE_SERVICE_REQUEST may be written as NL_ACT_SERV_REQ.

3.8.1 Factors which control the design of Service Primitives

It is quite evident by now that the real world protocol is far more complicated than the one defined by the protocol specification. The protocol specification provides only the peer-to-peer interactions for any layer, and barely touches the problem of inter-layer services. However for peer-to-peer interactions to be exchanged, messages should flow hierarchically between adjacent layers in any system within the protocol stack. A useful analysis of factors in development of quality of service specifications and current state of technology to support it, is given in [TeNg90].

Designing service primitives depends upon:

1. Generic factors arising from the OSI service concept.
2. Services provided by the underlying server layer.

3. Services the protocol under implementation ought to provide to its user, which in turn depend upon the peer-to-peer services.
4. Global behavior of the entire protocol stack under development.
5. The environment infra-structure which exists for an organization.

3.8.1.1 Designing Service Primitives from the OSI Concept

Although the number of primitives exchanged between any two layers depends upon the protocols supported by those two layers, there are some common primitives which are always exchanged between them irrespective of the protocols of the adjacent layers, and these are discussed below.

These OSI Service Specification Standards promote the concept of the Service Access Point (SAP) between adjacent layers. The Service Access Point can be viewed as an abstract point of interaction for communication between any two adjacent OSI layers. As an example, assume multiple users of a provider layer. Each user layer will therefore have to be identified to the provider layer.The provider layer will set aside a control block for each of its users because the states and operational environment of each of its users will differ and therefore should not be mixed. The user is identified by sending a specific integer or a specific pointer (address location) as the SAPI to the provider. The provider will now be able to identify each user by this SAPI. Hence, all communications between the layers should contain the SAPI, and the processes should check the SAPI before responding to an incoming primitive, in order to avoid any programming error related to identifying multiple users.

This concept of Service Access Point Identifier will be described as the Connection End Point Identifier in ISDN. The SAPI itself has a Network to End User or End User to End User significance in ISDN.

It is now possible to introduce four OSI concept-related service primitives generic to any OSI protocol:

1. The Service Access Point will come into existence through the exchange of a generic primitive called Identifier_ACTIVATE_SAP_type or Identifier_ACT_SAP_type.
2. Similarly a primitive to deactivate the Service Access Point is needed.
3. The above two primitives will be responded to or confirmed.

If the provider were to invoke the server first, as a result of the provider receiving a peer frame, the provider layer will identify itself with a Provider_SAP in its indication to the user layer, because the provider is not yet aware of the Service Access Point of the

higher layer (because the user did not ask for the provider services). As a result, when the user layer responds to this provider layer's indication, the user layer will send its User_SAP identifier along with the provider_SAP identifier. The provider can correlate using both the identifiers. This technique avoids the service primitive collision problem identified in [Cour87]. There are other comparatively complex solutions to the primitive collision problem such as in [Ansa83] [AyCo81].

3.8.1.2 Designing Service Primitives based on Underlying Services

The design approach suggested in this thesis is well suited to the design of lower layers, namely the network layer, the data link layer and the physical layer. For the sake of clarity, the underlying service concepts are applied to the design of the network layer.

When any layer is designed, obviously we must know, the service this layer can expect from the underlying layer Because the underlying layer is a data link layer, the following services can be expected according to the ISO LAPB[CCIT5] or LAPD [CCIT2] Layer Two specifications:

1. The data link layer permits the recognition of frames transmitted as a sequence of bits with the assistance of frame delimiting, alignment and transparency bit sequences.
2. It provides sequential control to maintain the order of frames across a data link connection.
3. Transmission, format and operational errors are detected on a data link.
4. Recovery from the above detected errors.

A detailed state machine representation of the X.25 Data Link (LAPB) Connection Establishment, Disconnection and Information Transfer Phases is given in [Kanu86].

It is not necessary for the network layer to be aware of all the above services. In fact, many of the functions of the underlying layers will be transparent to the higher layers: retention of service transparency at different layers of the OSI architecture is to be encouraged. The question then is: What are the visible services which the higher layer can expect from the underlying layer?.

Careful study of the specification of the server layer, or interaction with the designers of the server layer, will reveal the following factors for the LAPB data link

layers. This is the minimum set of service primitives which can be designed to extract the LAPB services:

1. Before data transfer can begin, data link connection should be established between the peer data link entities through exchange of certain layer 2 frames (handshake). For layer 2 LAPB frames, this handshake is achieved by the exchange of SABME and UA frames between the peer entities. The entity wishing to start the handshake will first send the SABME frame. A means should be provided such that the user of data link services can request the data link layer to start the data link connection establishment procedures. It can be done by providing a service primitive from the network layer to the data link layer. This primitive is DL_CONNECT_REQ. Details will not be given here on formatting of the bit pattern of the LAPB frames themselves, they should be transparent to the network layer.
2. It is important to understand the procedural behavior of the data link peer-to-peer establishment. For example, in LAPD, the data link layer will start a timer T1 after it sends a SABME to the remote. The SABME itself is as a result of DL_CONNECT_REQ. If LAPB does not receive a UA from its peer within T1, then a SABME will be sent again to the peer. This procedure will repeat N1 times. After N1 attempts have failed, a confirmation is sent to the higher layer with appropriate return code. The return code will indicate the link establishment failed after N1 attempts. Sometimes, the underlying parameters for a service, such as N1 and T1 have a range which is according to standards. Therefore the user of the entire stack may provide the above parameters to the network layer through NL_CONNECT_REQ which in turn is sent to the data link layer in DL_CONNECT_REQ. The pictorial representation of the data link establishment is shown in figure 3.7.
3. Similarly, when the peer data link layer sends a SABME first, the local user of data link services should be informed by the data link layer. This can be done with a 'DL_CONNECT_IND'.
4. The underlying service should end if either one of the peer data link layers desire. If the local user of the data link layer wishes to terminate the LAPB services, he can do so through a primitive called DL_DISC_REQ. When the termination procedure is successful, the underlying LAPB will send a DL_DISC_CNF primitive with the appropriate completion code.This primitive will force LAPB to send a DISC frame with the appropriate frame bits. On the other hand, if the local LAPB receives a DISC frame from the peer, it will send a DL_DISC_IND to its user and a UA frame

to its peer. The pictorial representation of the data link disconnection is shown in figure 3.7.

5. The network layer now knows how to establish and terminate the underlying service, it should also be able to exchange data with the peer. The network layer can send data when the underlying data link layer has established the data link. To send data it needs a primitive, which is called DL_DATA_REQ. This primitive will carry data to the user which will be sent by the underlying LAPB as an I frame to the peer. When the I frame is successfully transmitted, this network layer will get a DL_DATA_CNF from the underlying data link layer, with a successful completion code.
6. Aside from the design of the above common primitives for the establishment, data exchange and termination phases of underlying protocol (LAPB), the LAPB protocol itself will have service requirements on its underlying layer, even before the establishment phase can begin. These requirements may be, the underlying physical layer of the LAPB protocol needs to exchange certain end to end signals for synchronization at hardware level. For LAPB to get services from its underlying physical layer, LAPB should send a service primitive to underlying service provider(physical layer). This LAPB request in turn is invoked by having a primitive between the network layer and the data link layer. This primitive is discussed and named in the next section.

3.8.1.3 Designing Higher Layer Service Primitives

The design of higher layer primitives for a N layer protocol depends upon two factors:

1. Design of User Primitives based upon the N layer Protocol service alone.

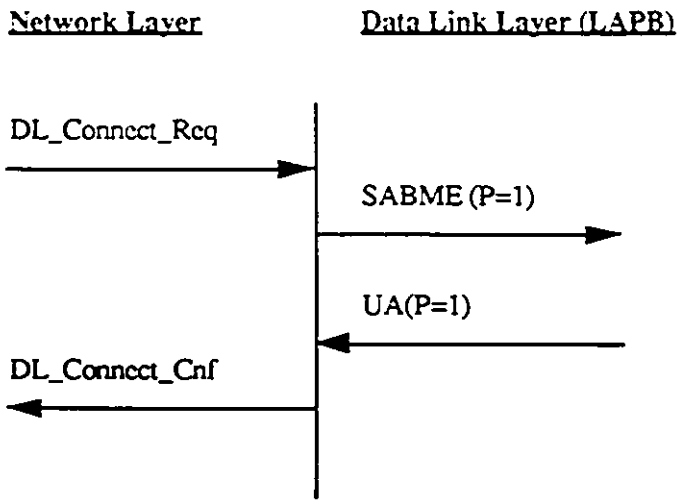
2. Design of the User Primitives based upon the underlying service concept for the entire stack from the physical layer to the layer under development.

The design of primitives based upon the first point will be discussed in chapter 6 with an example of the design of ISDN Q.931. The design of primitives based upon point 2 will be explained next.

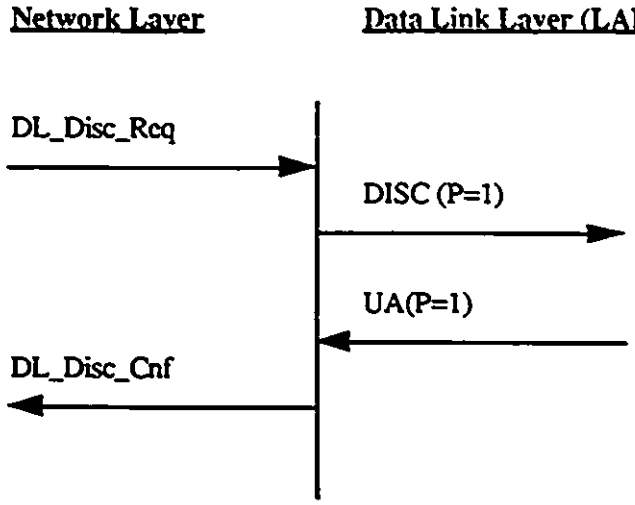
Just as we need to know the services provided by the underlying layer for the design of the underlying service primitives, we should likewise design the primitives for the services the user can expect from the layer under development. For clarity of understanding, the protocol under development is considered as the network layer. The

user primitives of the network stack are directly dependent on the global environment, therefore this section will cover points 3 and 4 of section 3.8.1.

1. Aside from the services which are specific to the network layer protocol under development, the user of the network layer should be able to tell the network layer to start providing the services. The user can do so with a primitive called NL_ACTIVATE_SERVICE_REQ. This is also essential to the network layer, so that it can send a DL_ACTIVATE_SAP_REQ (triggered as a result of NL_ACTIVATE_SERVICE_REQ) to the underlying LAPB, which in turn will send a request service primitive to the physical layer. The physical layer will now synchronize with its peer layer.
2. Once the services from a network layer are able to be provided, which is when the user of the network layer has received a successful NL_ACTIVATE_SERVICE_CNF, the services of the network layer can be invoked. As an example of the design of the network layer services, the design of the ISDN network layer will be covered in detail in chapter 6.
3. When the user of the network layer plans to terminate its services, he can do so through a primitive NL_DEACTIVATE_SERVICE_REQ, which in turn will force the network layer to perform peer-to-peer deactivation at the Network Layer level. Once peer-to-peer deactivation is performed at the Network Layer, it will issue a DL_DISC_REQ to the data link layer. On receiving DL_DISC_REQ, the data link layer will perform peer-to-peer termination at its level and then the complement of NL_ACTIVATE_SERVICE REQ will be performed at the physical layer to break the end to end physical layer synchronization. The confirm primitives will travel back with the appropriate completion codes.



Data Link Establishment Procedure



Data Link Disconnection Procedure

Figure 3.7 Data Link Connection And Disconnection Phases

3.8.1.4 Designing of Service Primitives from Environment Infrastructure

In section 3.8.2 the merging of service primitives, designed in section 3.8.1.2 and 3.8.1.3, is explained. This can be done in a number of ways. Often the final arbiter for the choice made is what had been done previously within an organization.

The amount of protocol code can be very extensive. The code size grows with the number of features provided in the protocol. To improve the debugging facility of the protocol code in a real-time environment, a number of error recording utilities (which may exist as independent processes) may be provided. Performance evaluation utilities are often provided as well. These utilities record protocol related information, such as the number of calls received, rejected, or accepted, the number of good and bad frames, the category of errors, etc.. These error recording utilities mainly expect a primitive from the protocol, in the event of an error in information, such as the type of frame or message received, the important parameters within the received PDU which are responsible for the error, etc.. Performance measurement utilities may provide primitives which will enable a protocol to change the threshold of acceptable and unacceptable service levels, etc.. The protocol will accordingly update the table controlling the peer-to-peer protocol and respond to the threshold request by sending a confirm. Thus, two-way primitives are required by the performance measurement utilities.

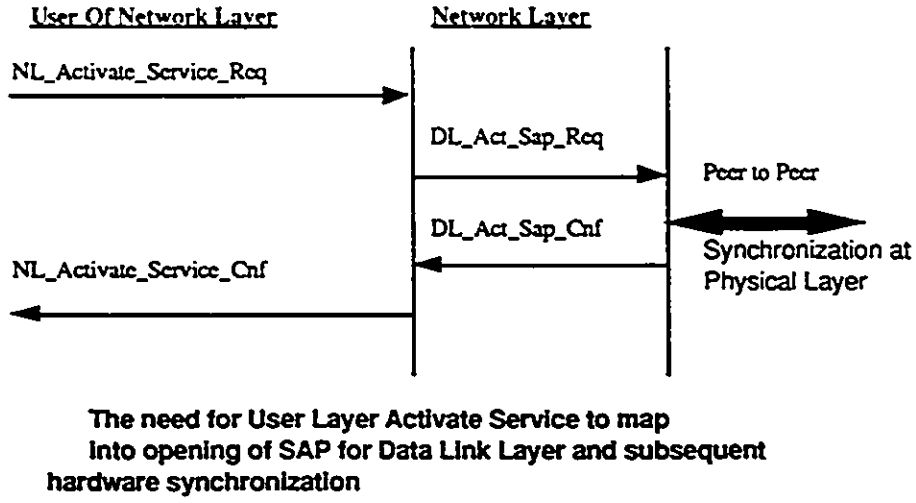
The design of the above primitives is not an international standard. Therefore, the previous practices and the existing environment consisting of tools for error and performance recording and changing, will determine the design of required primitives.

The details of local environment related primitives or error and performance related issues are not germain to this thesis.

3.8.2 Merging Underlying Services with the Services provided by the Protocol Under Development

It is now clear from section 3.8.1.2 that the services provided by the underlying layer from which were developed the interactions of figure 3.7, should be considered first. The next obvious question is *how the user of the layer under development commences the services of the layer under development*. To do this the primitives of section 3.8.1.3 (point 2) were introduced. In doing so, a global dependence on the underlying protocol stack was encountered, as a result of which the primitives of 3.8.1.3 (point 2) were mapped onto primitives to access the services of the underlying layer of the protocol under development. Therefore, the user of the protocol under development must follow a series of steps before the services of the protocol under development are accessed. An algorithm of the above steps is given in example 3.8.2.E1, and the combined interactions are shown in figure 3.8. For the algorithm to be developed, a variable is needed which indicates the inter-layer state of the protocol under development. (We are slowly getting into the realm of having a state machine for the inter-layer primitives.) There is no distinction(U or S) yet made in the development of the inter-layer state machine. In chapter 4, the process of enabling a

provider layer will be explained and will be merged with the above state machine. The resultant state machine will then have the distinction of the U and the S interface. The algorithm given in example 3.8.2.E1 merely represents the service activation process and does not show the error recovery or deactivation process.



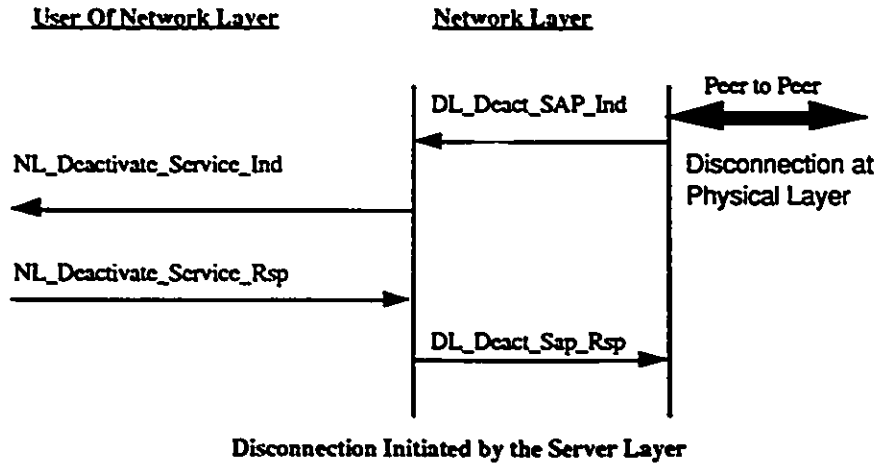


Figure 3.8 Merging Of Primitives

Example 3.8.2.E1 Main Entry Module to the Process

```
#include"Required_Files.h"          /* Include all system files  */
                                    /*Process or Protocol files */
                    /* this procedure is meant only to give a general idea of how a
                     higher layer primitive is mapped onto a lower layer
                     primitive—in this example during service activation of the
                     network layer . The procedure does not indicate all the
                     routines, variables and  constants */

call_state_machine_handler(struct SYS_ProcessCBType  Entity_CB_Ptr,
                                    int *internal_event,
                                    int  *return_code)
{
                    /*this procedure will determine the state machine to which
                     the received internal primitive should be given. It is assumed,
                     the layer was started_up previously and the received request
                     is for activating of the underlying service*/

                     determine_which_state_machine(..,&sm,  &primitive, ...);
                     /*sm is the local variable for determining of state machine
                     S_SM indicates the primitive is for S state machine     */
                     if (sm == S_SM)
                       {

                     call_S_stateMachine(from_global_pcb.S_State,
                                                    primitive, ...,
                                                    );
                       }
}

determine_which_state_machine(int *sm, int *primitive,
                                    ....);
{
                    .......
```

```
            /* the following 'if' statement ensures that the lower layer
            state is still INACTIVE, which means the lower layer was
            not previously activated by the Network Layer  in which
            case the primitive should go to lower layer state machine*/
            if (from_global_pcb.S_State == INACTIVE)
            {
                        *primitive = NL_Activate_Service_Req;
                        *sm = S_SM;
            }
}
call_call_S_stateMachine(
                              int InterLayerState,
                              int primitive_name
                            );

{
            ....
            Initialize_Activate_Provider_SAP(
                              DL_ACTIVATE_SAP_REQ,.....);
            Send_Provider_Primitive(.....);
            /* is explained in section 3.10 and  chapter 4.*/
            /* Update state of S state machine in global protocol control
            block to ACTIVATING */
}
```

Once the Service activation is requested by the User of Network Layer, and the Network Layer's request for activation of service from its Data Link Layer is satisfied as in figure 3.8, then the Network Layer will request for the underlying peer-to-peer data link connection to be established . Once this is done, the user of the Network Layer can use the

services rendered by the Network Layer. The use of these services will be discussed in chapter 6 with reference to ISDN Q.931.

In figure 3.8, when the user of the Network Layer requests activation of the Network Layer Service, the Network Layer issues a DL_ACTIVATE_SAP_REQ to the data link layer. The data link layer will issue a similar request to the physical layer and the physical layer will perform end to end synchronization if required. The confirms will then travel back. In the previous paragraph it was revealed that the Network Layer can then request that the data link layer establish the end to end link for layer 2. It is purely a local design issue whether to perform the physical layer synchronization and the data link establishment based upon the reception of NL_Activate_Service_Req by the Network Layer or have two separate network layer primitives. To keep the activation of the lower service transparent to the User of the Network Layer, the method followed in chapter 6 will merge the physical layer and data link layer synchronization with one request for activation of service at the Network Layer.

3.9 Designing Parameters for Service Primitives

There are two types of parameters for primitive design in the above section, namely the generic parameters and the parameters dependent upon the protocol under development or the services rendered by the entire protocol stack.

3.9.1 Basis for Design of Some Generic Parameters for Service Primitives

In order for the peer-to-peer data units to exchange data between the peer layers, the inter-layer service primitives must interact meaningfully accompanied with the data of the inter-layer and peer-to-peer data units.

Why Buffer Management Parameters are Required for Service Primitives:

Due to the presence of the peer-to-peer data, the inter-layer primitives now have the complexity of buffer management added to the task of appropriately mapping the inter-layer primitives based on the arrival of the peer-to-peer data in the purview of the global function of the layer. The global function of the layer indicates the history, mainly of the peer-to-peer communications. Buffer management includes functions which identify the type and size of buffer, and the nature of buffer (shared or individual) to be used for inter-layer communications.

Why Task and Entity Identifier Parameters are Required for Service Primitives:

Any N layer protocol specification may be implemented as multiple entities of that layer. For example, a N layer with two entities, one to send PDUs and another to receive PDUs. Another example is that the call control functions and management control of ISDN layers can be implemented as two different entities.

In the former case the two implemented entities are not peer-to-peer entities whereas they are in the latter case.

Any N layer entity can be implemented as an independent task(process), which in turn may consist of other sub-tasks(sub-processes). For example, the entity for sending data in the previous example may call a sub-task (sub-process) which is responsible for formatting and sending the data to the adjacent layers. The processes will communicate with entities in adjacent layers (tasks) through the operating system specific transfer calls.

The U, S and P state machines may be implemented as an independent entity, in which case the communication between the entities reduces to that of finite state machines. It is not necessary to implement different entities of a layer as individual sub-processes(entities), for example, the U, S and P state machines can be implemented as simple subroutines which communicate with each other through procedure calls with internal parameters or external variables. However, all the above subroutines will belong to a single N layer.

If a layer comprises multiple entities within a single task then the inter-entity communication primitives should identify each entity through a unique Entity Identifier..

The existence of multiple entities in any layer induces additional complexities rendering the protocol prone to error(s) and an excess of code.

The author suggests implementing any layer with a single entity in each one. This preference is supported by [Ahtia90]. In [Ahtia90] two problems are identified associated with multiple entities within a process: Managing Concurrent Connections and Scheduling Protocol entities within a single connection (process). These OSI implementational problems are discussed more thoroughly in [BoDe86] [GaHa88] [Svobo88].

Each entity will consist of multiple finite state machines. Therefore, each layer can be identified with a single task number and control will be passed to different state machines of the protocol through the local state machine identifier utility.

Alternative Method to implement Generic Parameters

The final buffer management design is based upon the operating system buffer management services and the design of the inter-layer primitives. This section should be

read in the context of section 3.10 and chapter IV for better comprehension. For example, the operating system buffer management may provide a pointer to the buffer which contains the inter-layer data as a parameter of the *send mail procedure* of the operating system. But the buffer pointer may be introduced as a part of the primitive structure itself, and the pointer to the primitive structure can now be passed as the buffer pointer. As long as the other layer knows how to interpret the incoming mail it does not matter how it is implemented. Similar arguments hold for the other generic parameters suggested above.

In this thesis the alternative method of implementation is followed.

3.9.2 Examples of Service Primitives with Generic Structures

The following are some of the important generic primitives and parameters required to implement an OSI protocol [HeRa78]. The function of the primitive and its parameters is explained briefly in the comments for each primitive in the following table.

```
        Primitive Function      The Higher Layer requests for the activation of the Layer
                                Under Implementation's(LUI) services.
        Primitive Structure
                Primitive Name  = NL_ACTIVATE_SERVICE_REQUEST
                CEI_Resolver    = Integer Type /* User = n or Provider = n+1 */
                                /* identifies service started by user or provider */
                CEI_value       = Pointer Type      /* it is actually the SAPI */

/*****************************************************************/
        Primitive Function      The LUI will confirm to the Higher Layer (HL) of the
Service Activation. The completion status actually depends upon
the activation of the service between the layer under
implementation and the provider layer.
        Primitive Structure
                Primitive Name  = NL_ACTIVATE_SERVICE_CONFIRM
                Return Code     =IntegerType
/*****************************************************************/
```

Primitive Function The Higher Layer requests the LUI for the deactivations of the services of the LUI.

Primitive Structure

Primitive Name = NL_DEACTIVATE_SERVICE_REQUEST
CEI_Resolver = Interger Type
CEI = Pointer Type

/**********/

Primitive Function The LUI will confirm the service deactivation to the HL. The completion status actually depends upon the deactivation of the service between the layer under implementation and the provider layer.

Primitive Structure

Primitive Name = NL_DEACTIVATE_SERVICE_CONFIRM
Return Code = Integer Type

**********/

Primitive Function The HL sends data to the LUI to be formatted as peer PDUs or directly to the peer in appropriate PDUs. This primitive is a generic one: the data enclosed within the primitive is irrelevant.

Primitive Structure

Primitive Name = DL_DATA_REQUEST
CEI = PointerType
Peer Data Pointer= DataBufferType
------> This pointer may alternatively be enclosed within the operating system mail handling service routine.

/**********/

Primitive Function The LUI will send a confirm to the HL depending upon whether it is a confirmed or unconfirmed peer-to-peer data exchange.

Primitive Structure

Primitive Name = DL_DATA_CONFIRM

```
            CEI                 = Pointer Type
            Peer Data Pointer= DataBufferType
            ReturnCode          = Integer Type
                                ------> If the confirm does not carry data with it then the
                                        Peer Data Pointer should be NULL.
/*****************************************************************/

      Primitive Function        Two other primitives for data exchange, namely
            DL_DATA_INDICATIONand DL_DATA_RESPONSE
            are required. They are similar to the above DL_DATA
            Request/Confirm.
```

3.10 Designing of Operating System Interactions

For a protocol to operate within a system, the operating system should provide facilities such as *Process Management*, *Inter-Process Communications* and *Timing operations* related to individual processes [TaVa86],[Amalu87]. This section reviews operating system concepts essential to the design of a protocol. The details of operating system services are further analyzed in chapter IV, on automating of protocol code generation. An analysis of the minimum operating system requirements is also performed in chapter IV.

I Process Management

The operating system should provide services to the protocol through certain easy to use *System Primitives* for Process Creation, Process Scheduling, Process Synchronization and System Error Handling.

A process is created when an instance of the program becomes executable. Each program will have its own *Static Area* where all the variables required for a meaningful protocol environment exist. The operating system itself will maintain a process control block(PCB) for the program. There may be multiple instances of the program, each of which is a process by itself. Certain modules of a program may be re-entrant, a re-entrant: module should not have any static variables within it. Control information such as the process name, and the program status word (PSW) are used to manage the execution of the processes.

The operating system itself will identify a process or a task through a *task_identifier*. A task-identifier is a pointer to a task or process control block which has important information related to the task. The structure of a process control block is particular to an operating system, but a typical PCB will have parameters similar to those shown in figure 3.9. If the amount of process control information is too large, some machines may break it up into secondary control blocks. The secondary control information will be accessible through the main PCB with the help of pointers in the primary PCB.

I.1 Process Creation

Any operating system which supports communications will provide facilities for dynamically creating processes using simple system function calls. For example, in a protocol network stack the network layer process may dynamically create the data link layer process and this process itself may create the physical layer process. These processes will communicate by message passing. For example, UNIX provides the function called "FORK", which enables a process to create another process. The communication between processes is through "PIPES" and "SOCKETS" [KeRi84, BELL83].

I.2 Process Scheduling

In any multi-process operating system, each process will compete for resources such as dynamic memory, CPU cycles, etc.. The scheduling of the processes is internal to the operating system. However, it is possible to provide priority to certain processes through operating system calls, so that they are given higher CPU cycles, etc..

I.3 Process Synchronization

Very often in multi-process programming there is a need to maintain the synchronization of processes, for example, when a section of the code in a process should not execute until some other process has executed certain functions. The process synchronization can be achieved through methods such as mutual exclusion, system semaphores, etc..

II Inter-Process Communication

Once multiple processes are created, communication will take place between them through the communication channels. These channels will need parameters such as the source and destination process identifiers for identifying the processes. Inter-process

communication will be through the exchange of specific operating system calls. The operating system may need information such as the amount of memory shared among processes, to establish a global communication mechanism between them. Inter-process communication may be uni-directional or bi-directional. The services of inter-process communication are also called the *dynamic services* in this thesis. The operating system provides these services through specific operating system calls.

III Timer Services

Timers are an essential part of the protocol specification, required to detect and recover from many kinds of protocol errors. Therefore the operating system should provide the timer services to the protocol. This is done through specific operating system routines for starting of timers, stopping of timers, cancelling of timers, etc.. Timer related system routines will provide a specific *timer identifier* during timer creation time, which will be used for stopping or cancelling a timer subsequently.

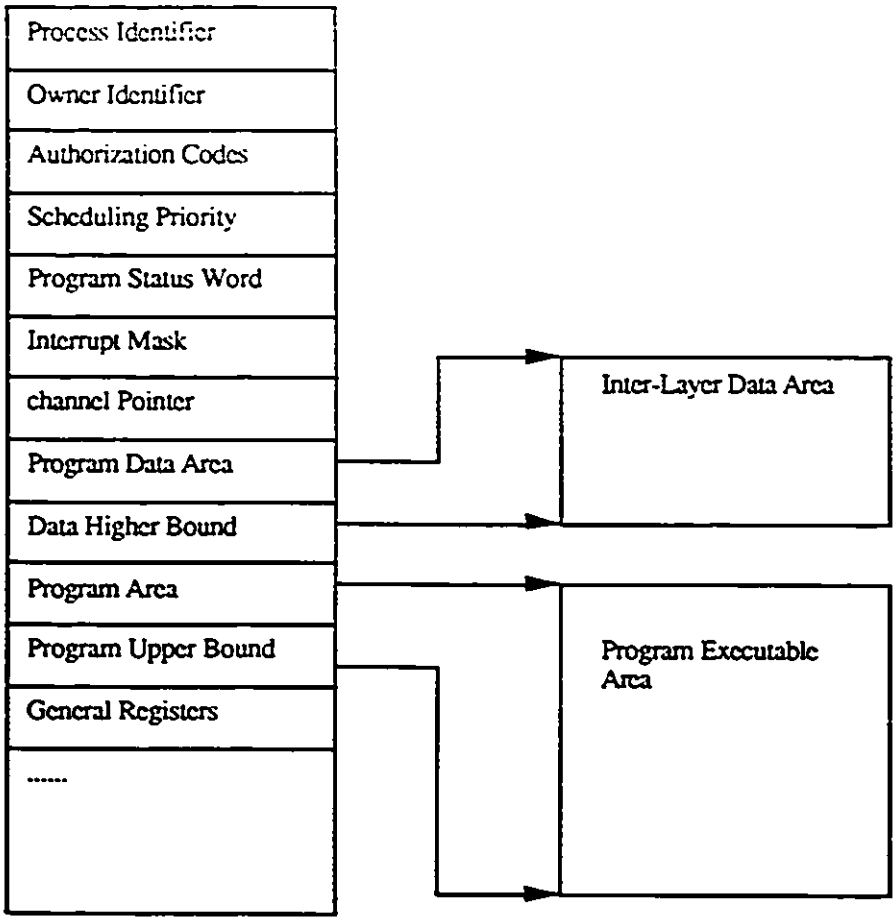


Figure 3.9 A Typical Process Control Block

The next chapter will make use of these system concepts for the definition of a standard system interface which will be used for the study of the feasibility of the generation of complete executable protocol code.

Chapter 4

4. Automating of Protocol Code Generation

The operating system interface is a major part of the design phase of any communication protocol. However, it is not emphasized in the protocol design literature. In this chapter we present an analysis of the literature concerning semi-automatic protocol implementation and formal specification area with respect to the operating system interface issues.

In a stepwise design and implementation of an open system protocol, the adoption of OSI architecture occupies the highest priority, followed by the design of the protocol primitives based on the OSI service specifications then enhancement of the set of these primitives to co-exist within a local environment and lastly the introduction of the system interactions of the protocol to be meaningful with respect to the system.

During the design process of a protocol, in order for it to operate within an environment, we must consider its interactions with the executing underlying system together with the protocol specification, the OSI service concept, and the practices of development prevalent in an organization. In the design process of a protocol the interpretation of the specification is unique, except for valid options within the specification itself. Services between the layers are a function of the protocol requirements of the adjacent logical layers and the environment. The variables of the design process are widely discussed in [BoSu80] [BoGo86] [Boch87]. Of the many variables which control the design process of a protocol, the system interface has attracted the least attention, even though it is integral to any protocol engineering process. A key breakthrough for automation has been the development of formal specification languages called Formal Description Techniques of FDTs. From FDTs it is possible to obtain a substantial amount of system-independent protocol code through automated methods [Boch80] [VoLa88]. However the system interface of a protocol which is a key aspect of a protocol design has so far not been standardized. A proposal for system interaction standardization can be

found in [AlFe90], much of the contents of which are included in section 4.3. A similar "smoothing approach to operating system interface" is given in [Koen90].

Though an availability of a standard system interface which can cater to any OSI protocol is desirable, it is not likely to be agreed upon easily by various manufacturers. Similarly, it was once difficult to standardize the inter-layer interface between adjacent OSI protocols.

Concerning the capturing of protocol interactions with the operating system in the design process, the following problems and suggestions should be noted [Boch87] [VoLa88] [SiCh89] [Koen90]:

1. Operating system interactions are not portable.
2. System interactions depend upon the underlying hardware and the operating system.
3. The user of the system interface requires a particular implementation and structure or style.
4. Lower layer protocols must interact with the underlying hardware, and the environment can be implemented by invoking a set of system dependent routines.
5. The machine dependent part of the specification varies from 25-50% of a complete protocol implementation, and is hard-coded in a semi-automatically generated protocol implementation.
6. Local implementation matters are considered as the major challenges in OSI implementation, yet OSI has no way to address issues such as control between layers, interfaces between layers and host system, and memory and buffer management.
7. Once the machine-dependent part of the protocol is implemented it can be used again for a different protocol running under the same operating system.

4.1 Extended Operating Systems and Protocols

In its final form, a protocol layer is nothing but a machine executable code which runs on a particular CPU. Like any other conventional program, the protocol executable code should have enough dynamic storage available during the life of the protocol. Depending upon the system on which the protocol runs, which may be a personal computer, a network controller, an intelligent work station, etc., the CPU may be located on an additional card in the extended system or the protocol may run on the main CPU on the mother board, as shown in Figure 4.1.

More and more of the systems manufactured today tend to have the extended slots into which the custom made protocol cards can be inserted. The CPU on these cards

normally has an operating system different from the main CPU. We call this operating system the *extended operating system*. There is now a need for a transport mechanism between the main CPU and the extended system.

The protocol is coded in conventional languages such as C or Pascal on the main system, but uses the operating system interface of the extended system. That is to say, the protocol code must be compiled with all the libraries, such as the extended system operating system libraries, other supporting utilities, etc.. The generated object code should now be linked such that it runs on the extended system.

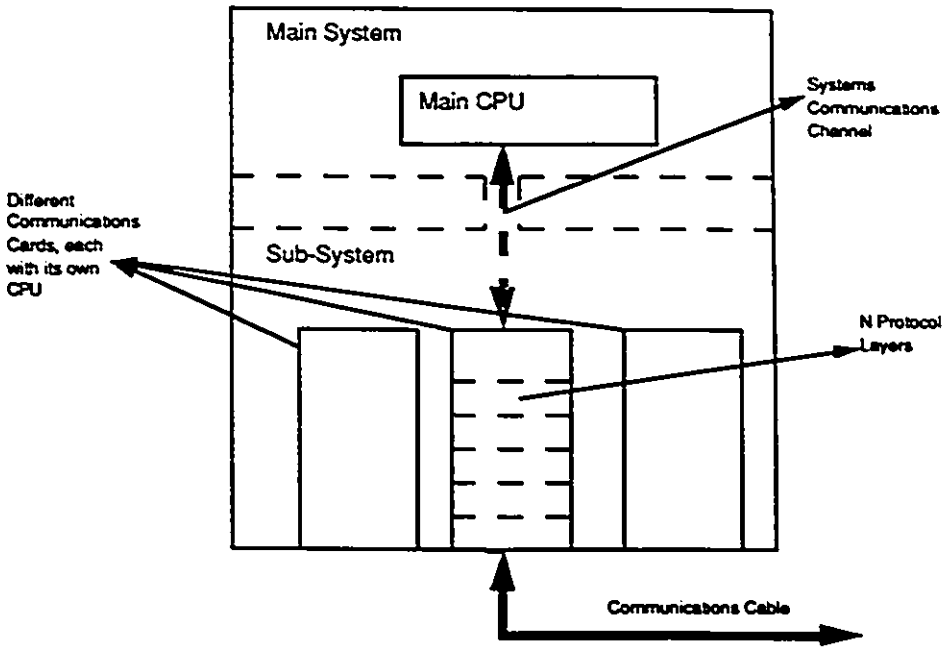


Figure 4.1 A typical System and Sub-system

4.2 Effect of the Operating System on design of protocols

The operating system interactions of the protocol depend upon the extended operating system running on the extended system CPU. Operating systems for OSI protocols usually have a father and son relationship, to suit the user and provider layer concepts of OSI. The operating systems for communications protocols are usually of the multi-tasking type.

A task normally owns and schedules resources such as hardware or storage for data transmission, reception etc., The tasks are known to the operating system. The services provided by the operating system to the tasks vary widely from system to system.

The operating system is responsible for starting a task, and schedules the resources for different tasks based on different scheduling algorithms, or based on simple round robin time sharing.

Each protocol layer may be implemented as a task, or multiple layers may be present in a single task. In the latter case, each layer may be called an *instance* or an *entity* in the task. A task usually starts up different layer executable codes (entity) and therefore has information such as the layer identifier for each layer or an entity identifier for each entity. Layers which are either tasks or entities within the sub-system comprise the *protocol stack*. When the main task of the protocol stack controlling all the entities is enabled by the application, this task can identify each layer with a *layer_identifier*.

4.3 Service offered by the Extended operating system to the Protocol

The protocol layers in general make use of the following services related to a task in an extended operating system environment [HeRa78]:

Startup lower layers

The protocol stack is normally started by an application layer in the main system. The application layer enables the uppermost layer of the protocol on the extended system with an enable_request. During the enable period, the user of the layer provides the user options to the provider layer. As this layer requires the services of its lower layer, it enables the lower layer and this process continues until each layer is enabled. The provider entity is not enabled if the user options are invalid, and it will send a startup_confirm with the specific error code. The error code is propagated in the respective startup confirm until it reaches the user in the main system.

It is not necessary that the user startup-request command should descend from the top layer to the bottom layers, but this is merely a design convention. It is possible for the physical layer to be start-up first, which results in a hardware synchronization with the network and ascent of startup indications. If the higher layers start-up successfully, the responses will travel backwards; in the event of failure of start-ups appropriate error codes will be sent. During the enable period, each layer will also acquire some system resources for its dynamic execution. They also acquire the identifier for shared resources, if any. If this initialization fails, a startup_confirm or startup_response is sent with error code for the initialization failure.

Shutdown lower layers

The shut down or disable process is normally initiated by the user by issuing a command to the application layer in the main system. The application layer will send the closedown_request to the topmost protocol layer on the extended system through the transport mechanism. Every layer of the task will now issue a closedown_request to its provider entity. This process is continued until every entity is disabled on the extended system and until the shutdown request from the main system is confirmed with success. During the disable period, every layer will cancel all timers and free all resources.

Mail Handling

The task managing a group of entities will maintain a specific storage called the *Entity Block*. Information in the entity block is shared between any entity and the task whenever there is a mail transmission. For example, the entity block may have information such as the origination entity identifier of the mail, the target entity identifier, request/confirm/indication/response identifier etc. Once the mail is delivered or received, this entity block is updated by the task.

Storage Handling

There may be different concepts for data transmission between layers. In general, when data is to be transmitted from one layer to another, the source layer should first acquire a buffer from the transmit buffer area by issuing a system command, for example get_buffer. Get_Buffer may return a pointer after successfully completing the command. The source layer will initialize this buffer with the data to be transmitted before calling another system with the call 'send_mail' to transmit this buffer.

Timer Handling

Every communication protocol will have certain timer-based peer-to-peer PDUs, mainly to recover from loss of PDUs in the communication channel. There are many other advantages to having a timer-driven protocol. To support timers within the protocol, the protocol will request timer-related services from the system interface. The main timer-related services are that, the protocol should be able to get a timer from the system, start a timer, stop a timer and cancel a timer.

If the above buffers are of a certain size, and if the data transmitted varies from primitive to primitive, then there may be different areas from which the data buffers could be accessed before transmission. For example, the primitive pool(an area containing buffers) contains primitive_buffers for primitive header transmission and the data_pool contains data_buffers for data transmission.

Releasing of buffers depends upon the operating system or the task handler implementation. Depending upon the operating system, the buffers may be released either by the protocol entity or by the task itself.

There may be another form of storage acquired simply for local data processing, and not for transmission of data. It can be acquired by making another operating system call such as 'get_local_storage' which returns a pointer to an area of the requested size. It is similar to the *malloc* function call of C.

4.4 Operating System Factors for automated protocol implementation

The above topic depicts the complexities involved in the startup, shutdown, mail handling and storage handling aspects of any protocol operation. These common functions may be handled differently on different systems running under varying operating systems.

To generate protocol code semi-automatically, the above functions must be generalized, so the exact sequence of invoking the operating system services and the parameters for such a communication must be outlined to the protocol developers before the protocol code is designed.

This thesis proposes an idea for a standard operating system interface which eliminates the operating system dependence by protocol developers. Instead system programmers will assist them by developing a standard system interface designed according to the generic protocol requirements.

4.4.1 Examples of Standard System Interface Calls with some Generic Parameters

The intention of the thesis is to convey the general idea of having a standard system interface, such that the automation of production of the entire protocol code is feasible. Therefore, only a few examples of the standard system interface services explained in section 4.3 will be presented. The standard system interface will provide all the services of

any multi-tasking protocol as well as services such as Flow control for the different protocol layers.

Services Related to Starting -up of the Lower Layers

For startup_lower_layer services to be provided by the operating system, the user layer should invoke a system call such as the following C procedure call. Each parameter may be very complex and it is outside the scope of this thesis to discuss design issues pertaining to implementing the Standard System Interface. We merely want to point out the generic concepts in such a Standard System Interface.

```
SYS_STARTUP_PROVIDER_REQUEST(&ChannelIdentifier
                             &RequestIdentifier,
                             UserLayerIdentifier,
                             ProviderLayerIdentifier,
                             UserConfigurationParameters);
```

Explanation of Design and Parameters Used

The parameters themselves are particular to a local environment, because the concept of Connection Identifier Request Identifier, Layer Identifier, User Configuration Parameter, etc., may be different. Moreover, these parameters need not all be exchanged through one operating system service call as above. For example, the user of a provider layer knows what to expect from the provider layer, and may therefore change certain variables used by the provider layer in order to get the expected service. This change of parameters is made by passing a pointer to a location where in the User Configurable Parameters are saved: the provider will get them from this location. The pointer to such a User Configurable Parameters may be passed during the invocation of the lower layer in the same function call or may be passed as a separate function call. Similarly, Channel Identifier in the above example simply identifies a connection between any two adjacent layers once the provider is enabled. Therefore, once the provider is enabled successfully, the user layer need not send the User Layer Identifier or the Provider Layer Identifier each time it sends or receives an inter-layer communication primitive. However, this could change if designed differently (for example, if there is no concept of Channel Identifier, then the communicating layers should always identify the layer identifiers during every interaction).

The layer identifiers are required initially to identify the program area wherein lies the executable code for that layer, as shown by the Program Area and Program Upper Bound pointers in Figure 3.9.

The Request Identifier is a generic identifier, promoted as a result of OSI's service primitive types. For example, if multiple requests are sent out, there must be a way to correlate them when the confirms come back. To achieve this the operating system will return a Request Identifier whenever a request is sent out. Later when the confirm comes back, the sender of the previous request will check for the Request Identifier within the received confirm. Thus, the correlation can be achieved enabling a layer to communicate with adjacent layers independent of the order of interactions.

Parameters may be added or deleted depending upon the local acceptance. Similarly, the confirm for the above reques- will be as shown below:

```
SYS_STARTUP_PROVIDER_CONFIRM(ReturnCode,
                             RequestIdentifier);
```

If the layer could not be started, then the reason for this is returned as the return_code. The Request Identifier will help the sender of SYS_STARTUP_PROVIDER_REQUEST to correlate the confirm with the request.

Services Related to Shutting Down the Lower Layers

Similar to the startup lower layer request, there will be a system interface call to shut down lower layers.For example,

```
SYS_SHUTDOWN_PROVIDER_REQUEST(ChannelIdentifier, ....);
```

```
SYS_SHUTDOWN_PROVIDER_CONFIRM(ReturnCode,
                              RequestIdentifier);
```

Services Related to Mail Handling

```
SYS_SEND_MAIL_REQUEST(ChannelIdentifier,
                      PrimitivePointer,
                      PrimitiveDataPointer,
                      &RequestIdentifier);
```

The Channel Identifier shows the layer to which the primitive and the data associated with this primitive must be delivered. The Primitive Pointer points to the buffers where the inter-layer primitive information is initialized. The Request Identifier is required to correlate this request with the confirm. The system call given above includes the PrimitiveDataPointer, therefore the PrimitivePointer should not include a data pointer .

```
SYS_SEND_MAIL_CONFIRM(PrimitivePointer,
                      PrimitiveDataPointer,
                      RequestIdentifier);
```

The Primitive Pointer itself points to the primitive information template so the primitive may be sent. The Primitive Data Pointer points to the data buffer where the information to be sent with the primitive is initialized. The Request Identifier which was sent with the previous SYS_SEND_MAIL_REQUEST will be sent back with this confirm.

Services Related to Buffer Handling

For inter-layer primitives to be sent to the adjacent layers, the information on the type of primitive and the data to be sent should be initialized in certain buffers. Therefore, the layer which wants to send the mail should request system storage, called the buffers. Because the primitive structure size may be much smaller than the data to be sent, and the data itself may also be different for different types of service primitives, we therefore need a parameter in the system call such that a buffer of a certain type or size can be obtained.

Similarly, since the buffers are allocated dynamically, in order not to exhaust the system memory the buffers should be released once it is no longer being used. The buffers are acquired by a layer before they are sent to the adjacent layer. Once the mail is delivered to the adjacent layer, a design decision should be made whether the buffer

will be released by the target layer, by the source layer or by the operating system itself. A mechanism should be provided to release the buffer in the manner decided. Also, the buffers may come from an area of memory shared between the layers. There must be a mechanism to inform the system interface if buffers are shared.

Two sample system calls related to the buffer handling services of the system interface are shown below:

```
SYS_GET_BUFFER(BufferSize,
               &BufferPointer,
               &ReturnCode);
```

The protocol will tell the system interface the size of the buffer it needs through the BufferSize parameter. The system will return the pointer to the allocated buffer through the BufferPointer. In case there is a problem allocating the requested buffer, the Return Code will identify the type of error.

The following procedure will enable a protocol layer to release the buffer back to the operating system.

```
SYS_RETURN_BUFFER(BufferPointer,
                  &ReturnCode);
```

Services Related to Timer Handling

A protocol may need to have multiple timers running. Therefore, when the protocol requests a timer, the system should provide a timer identifier. The subsequent timer-related requests, such as start a timer, stop a timer, cancel a timer etc., will need the timer identifier as a parameter of the system call.

An example for Get_Timer and Start_Timer is shown below. The key parameter, the Timer Identifier, is also shown. However, other parameters of interest based upon the local requirements may be designed and coded. An example of a local requirement would be the need of each layer to start its own timers if there are multiple layers. To facilitate the system interface in keeping track of Timers on the basis of an individual layer, the Source Layer Identifier may be provided.

```
SYS_GET_TIMER(&TimerIdentifier,...);
```

SYS_START_TIMER(TimerIdentifier,...);

Other Protocol Synchronization Services

Any layer can flood the communication channel of its adjacent layers by sending an infinite number of primitives. This potential problem should be controlled by the standard interface (or operating system) by ascertaining from the protocol layers (before invoking a layer) the number of primitives of each type which can be sent to and received from all its adjacent layers. In the event a layer tries to send more than a pre-defined number of primitives, actions will be taken as designed. As a result the protocol designer is now free of the flow control mechanism and will concentrate only on the design of the protocol [HeRa78].

This standard interface is now responsible for managing the queues between protocol layers. In the queue itself, there may be different types of primitives from adjacent layers and entities. The design of queue handling between adjacent layers or entities within a layer should incorporate queuing and dequeing priority schemes. The protocol designer should be aware of the priorities in handling primitives within the channel. Thus, the problems of Collision of Service Primitives and Backpressure Flow Control [Cour87] will be resolved in the standard system interface by mechanisms such as those given in [Ansa83] and [AyCo81].

4.5 Merging System Services into Inter-layer State Machines

Of the above services, only the Startup and Shutdown Services of the operating system will be merged with the U state machine of section 3.7.1 (this is due to our selection of services for the creation of a state machine to promote atomicity in Estelle modules). Any protocol is merely a executable piece of code located in certain memory location. The higher layer should invoke this code by making use of *Startup System Service* command. Likewise, to power down lower layers, the higher layer will make use of *Shutdown system service* command.

The entire process of providing services to the user of Network Layer is now a sequence of comprehensive steps.

1. The Network Layer will be invoked by receiving the SYS_STARTUP_PROVIDER_REQUEST (shown as SYS_STARTUP_REQ in diagram 4.2 to reduce the length of the name) from the User Layer (receptions shown by a + sign in the diagram). As a result, the Network Layer will invoke its

provider layer with the help of SYS_STARTUP_PROVIDER_REQUEST (in the diagram it is shown as DL_STARTUP_REQ to explicitly indicate the starting up of data link layer, and the - sign indicates it is a send transition). On successful startup of all the provider layers, the confirms will travel back with the use of the system service SYS_STARTUP_PROVIDER_CONFIRM which is triggered as a result of reception of a successful DL_STARTUP_CNF.

2. Once all the provider layers are enabled, the User Layer should first startup the service of the Network Layer by issuing the command NL_ACTIVATE_SERVICE_REQUEST (NL_ACT_SERV_REQ). As a result of this activate service request, the Network Layer will activate the Service Access Point with its provider layer by sending a DL_ACTIVATE_SAP_REQ. On receiving of the DL_ACTIVATE_SAP_CNF, the Network Layer will request the data link layer to establish a peer-to-peer data link.
3. OSI recommends that the underlying services should be transparent to the upper layers. As a result of NL_ACTIVATE_SERVICE_REQ the Network Layer not only activates the SAP with the lower layer but also requests the underlying data link layer to establish a peer-to-peer data link layer connection, by sending DL_CONNECT_REQ.
4. When the peer-to-peer data link connection is successfully established, the data link layer will send a DL_CONNECT_CNF to the Network Layer.
5. The Network Layer will now send a NL_ACTIVATE_SERVICE_CNF (NL_ACT_SERV_CNF) to the user.
6. At this point, the User Layer can make use of specific Network Layer services by issuing the permitted primitives, as explained in chapter VI with reference to Q.931 protocol.
7. Similar procedural steps are required to deactivate the service and then shut down the entire underlying protocol stack from the User Layer's point of view.

The above procedure will be divided into two state machines, the U state machine and the S state machine. The events coming into the N layer protocol can reach the S state machine only if the U state machine is enabled. Thus, the S state machine is dependent upon the U state machine. Likewise the incoming peer interactions can reach the P State Machine only if the S state machine is in the ESTABLISHED state.The details of U and S state machines are shown in figure 4.2.

In this chapter the concept of a standard system interface was presented and the merging of start-up and shut-down services offered by such an interface into the service

specifications was proposed. The resulting service specifications yield finite state machines which serve the adjacent layers for specific services.

The inter-layer state machines generated above should be merged with the services offered by the protocol under development. The protocol under development is ISDN Q.931. The next chapter is devoted to understanding of the global picture of ISDN and existence of Q.931 within it.

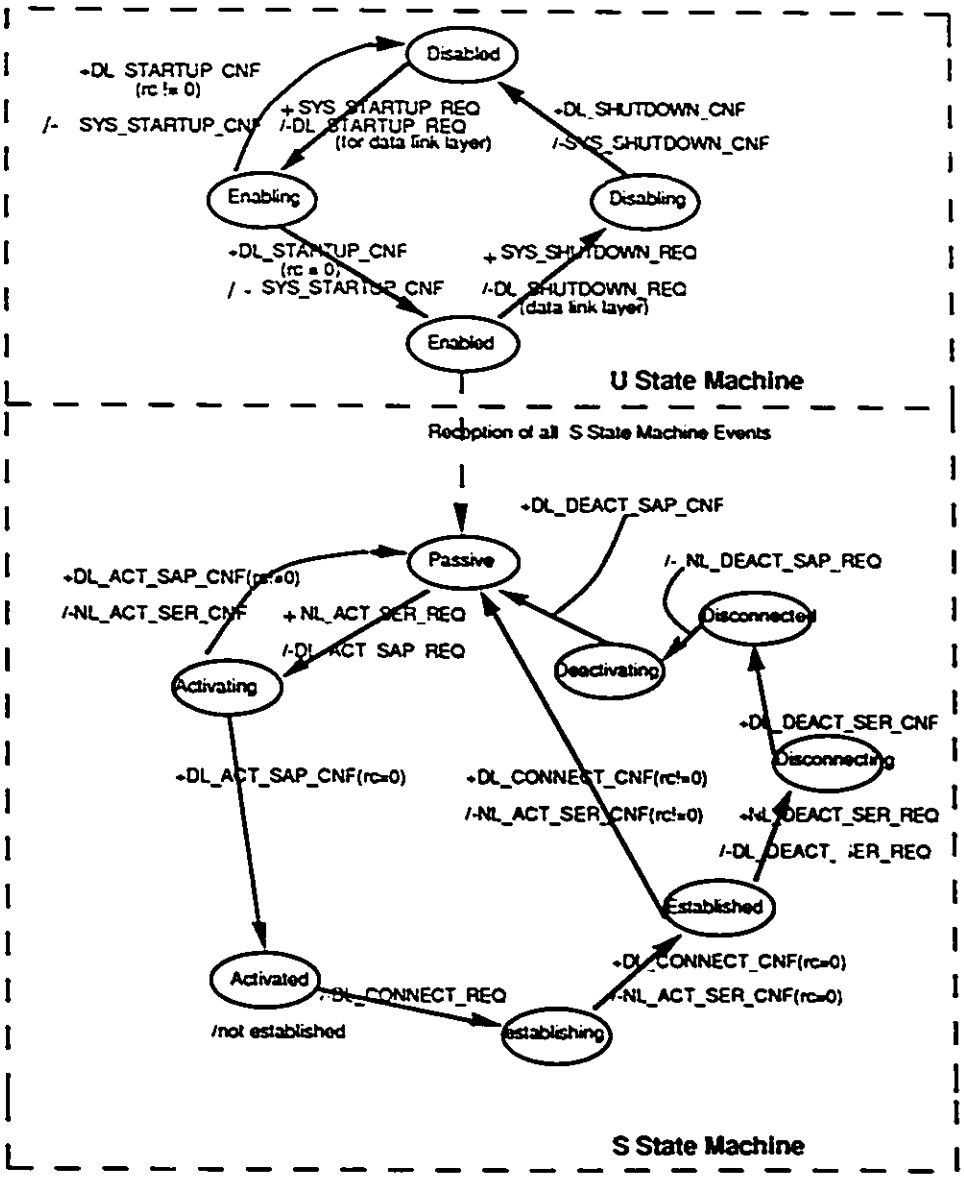


Figure 4.2 U And S State Machines

Chapter 5

5. Overview of Relevant ISDN Concepts

ISDN stands for Integrated Services Digital Network. As the name suggests the objective of ISDN is to *integrate* all the existing and future telecommunication services into one network and a few well defined standard interfaces to the network.

The existing telecommunication networks are heterogeneous and are dedicated to a single service, which in itself may have many features. For example there are telephone networks for voice transmission services, packet switched/circuit switched private and public networks for data transmission services etc. As an example for features of a service, the voice services may have Call-Forwarding, Call-Blocking features etc. Any user who wants to make use of multiple services mentioned above, should support his system with different hardware and software, has to undergo complicated negotiation with the operating companies for the services to be made accessible through multiple physical access points. The user will eventually end up with numerous physically identifiable communication systems, communication cards, associated hardware accessories and connecting cables and wires as well as numerous software or microcode packages. As a result the system, software and microcode installation will require comprehension of enormous publication material for configuring the system depending upon the customer's requirements.

The above outlined facts cannot be overlooked. In reality large organizations depend on many telecommunication services for their day to day activities, as a result of which they require constant maintenance of the above identified hardware and software/microcode support, eventually leading to a lot of misery. The user will run into a highly complicated, expensive and unreliable communications setup.

In order to simplify the communications services, ISDN has evolved. ISDN is not an altogether new technology, it is just a collection of all previous communication services such as Telex, Voice Telephone, Data transmission etc. into a one single physical entity. Because of the availability of larger band widths, ISDN is paving way for new services

such as multiple calls, video and image transmission etc. through the same set of interfaces and networks.

There are numerous public and private networks around the world today. each network has its own idiosyncrasies. Thus exchanging of information between different networks is highly complicated.The goal of ISDN is to unify the diverse networks into a single network which can cater to all the possible services at certain well defined user interfaces.

ISDN networks are based on a 64KBPS data rate and are intended to support voice facilities, existing data services, and a large number of new and extended facilities. Requests for these facilities is made on a common signalling channel with a common set of signalling protocols. These common signalling protocols will continue to grow as new services are added to ISDN. Thus high band widths for large applications can be negotiated through low speed common signalling channel of ISDN.

Following are some of the services which will make use of ISDN [Potter85]:

Existing Telephone Services
Circuit Switched Data
Packet Switched Data
Electronic Mail
Videotex [ChSa89]
Teletex [RoCa81]
Facsimile
Remote Sensing Services
Voice and Data Supplementary Services

A comprehensible and complete explanation on general aspects of ISDN and major protocol issues of ISDN are given in the special issue [Select86].

5.1 ISDN User-Network Interfaces

In order to support different user service requirements ISDN provides two major interfaces namely, the Basic Interface and the Primary Interface.

Basic Interface

It provides the user with two 64Kbits/sec B channels and a 16Kbits/sec D channel for a total of 144Kbit/s. However, the total band width provided by the

Basic Access Interface is 192Kbit/sec. The difference bits are used for network management purposes and the user cannot use them for information exchange.

The above band width is enough to meet the requirements of most individual users. It allows simultaneous use of voice, high-speed data, facsimile or image and several text and teleaction applications.The hardware base(together with software) for the above applications may be a single multi-function terminal or several terminals.

Primary Interface

Primary rate will support higher rate applications and is especially required to support terminals which make use of a PBX, LAN or controller such as NT2. Primary rate is divided into two categories, namely the North American Standard Primary and the European Standard Primary.

The North American Primary Interface will support 23B channels and one single 64Kbits/sec D channel. The European Primary will support 30B channels and one 64Kbits/sec D channel.

Other Interfaces

The applications may make use of certain fixed number of B channels grouped at a time. Thus a group of B channels may be accessed as H0 channels (384 Kbps), H11 channels (1536 Kbps in USA and 1920 Kbps in Europe) and H12 channels (1920 Kbps).

The overall specification of the basic and primary access interface is in CCITT I.420 and I.421 respectively and the physical layer of the basic and primary access interface is specified in I.430 and I.432 respectively. The D channel is specified for both basic and primary access interfaces in recommendation I.440 and I.441(layer 2) and I.450 and I.451 for layer 3.

5.2 ISDN Signalling

ISDN differs from other conventional protocols in its signalling method. ISDN has *distinctive* physical channels for signalling and transmission of data. Even though the D channel signalling and B channel data transmission takes place on the same pair of physical wires, by *distinctive* we mean the D channel and B channel bits always occupy a limited number of specific bits in the frames transmitted or received by the hardware.

The signalling which takes place on a channel which is different from the one on which the data is transmitted is called the *outband* signalling. The data exchange can take place on B channels according to different protocols which support data transmission such as X.25, Facsimile, Videotex etc., while the signalling is always in accordance with ISDN Q.931 protocols.

The data transmission protocols such as X.25, SNA will still be used in ISDN. Thus ISDN is nothing more than outband signalling for transmitting different data protocols. Since the common signalling is used for transmitting voice, data, text etc, in order for the same physical user-TE's to be used, it should off-course have the hardware support for the required service, eg., a handset should accompany the TE if voice signalling and voice data has to be exchanged, Fax hardware should be available in order for letters to be sent or received. X.25 protocol should be available in order for electronic messaging to be done etc. In this thesis, ISDN signalling protocol is used as an example. Thus ISDN implies ISDN signalling alone.

For certain access configurations, the D channel can also be used as data transfer channel during non signalling time period.

5.3 ISDN Communication Modes

The major communication modes supported by ISDN are

Circuit Switched Mode

Provides end to end digital connection at the transmission rate of the selected B channel. The channel may have transparent transmission of bits or coding may take place for specific services eg.,coding and decoding of voice into digital signals for telephony.

Packet Switched Mode

Packet user information can be sent over the B channels or the D channels. Packet data transfer protocols such as X.25 have to be employed in order to do so.

Semi Permanent or Permanent Connections

Some times an ISDN signalling may take a pre-defined permanent path through the ISDN networks.

5.4 Service Aspects of ISDN

Customers make use of features and services offered by telecommunications service providers through services supported by ISDN. The telecommunications services which can be accessed through ISDN are divided into two categories:

Bearer Services [Potter 85]

The user of ISDN (from layer 1 to 3) will negotiate and establish certain features based upon user ISDN implementation and network ISDN implementation support. This thesis will highlight the Bearer Services aspects of ISDN implementation.

Tele Services [Potter85]

The user applications perform certain data communication activity which make use of ISDN layer 1 to 3. The layers 4 to 7 which define the type of application which may run on the ISDN bearer services, together constitute the Teleservices of ISDN.

5.5 ISDN Access Points and Termination Devices

The access to an ISDN network is through a set of well defined interfaces namely R, S or T interfaces as shown in figure 5.1.

T Access Point

It is the minimal interface which should be provided to the user end. The user ISDN terminal should be able to access this point in order to communicate with the network.

S Access Point

It corresponds to the interface of the individual ISDN terminals.

R Access Point

It enables the non-ISDN terminals to use ISDN services. R interface may be provided in the form of external adapter which on the user side will accept the protocol in use by the user while on the network side it will look like an S access point.

The terminal equipment which the user can own in order to access the above ISDN reference points are TE1, NT1, NT2 and TA as shown in figure 5.1.

TE1(Terminal Equipment Type 1)

It is the user equipment which can access the ISDN network through S or T reference points. It may be provided in the form of a communication card which is inserted into the existing user terminal.

TE2(Terminal Equipment type 2)

It is a non ISDN terminal which can interface at the R reference point. These equipment usually follow X-series and V-series recommendations such as X.21, V.21 or V.28 etc.

TA(Terminal Adapter)

It permits a non-ISDN terminal(TE2) to access the ISDN user-network standard interface.It will support the protocol of the TE2 on its user end side and the ISDN protocol on its network side.

NT1(Network Termination 1)

It is the line termination which is provided to the user premises by the ISDN network vendors. NT1 is the minimum component essential to access the ISDN network services. NT1 will contain certain physical layer functions such as line maintenance, power feeding and other electrical characteristics. The reference point provided on the user side of NT1 is the T reference point.

NT2(Network Termination 2)

It has two access points. On the user side it provides S interface and on the network side it provides the T interface. NT2 may have functions of PBX's, LANs and controllers. On the user side it can support multiple TE's. Since there are multiple TE's who can talk to a single NT2 there should be some form of medium access to be shared between TE's. Passive bus configuration is the standard medium access for ISDN.

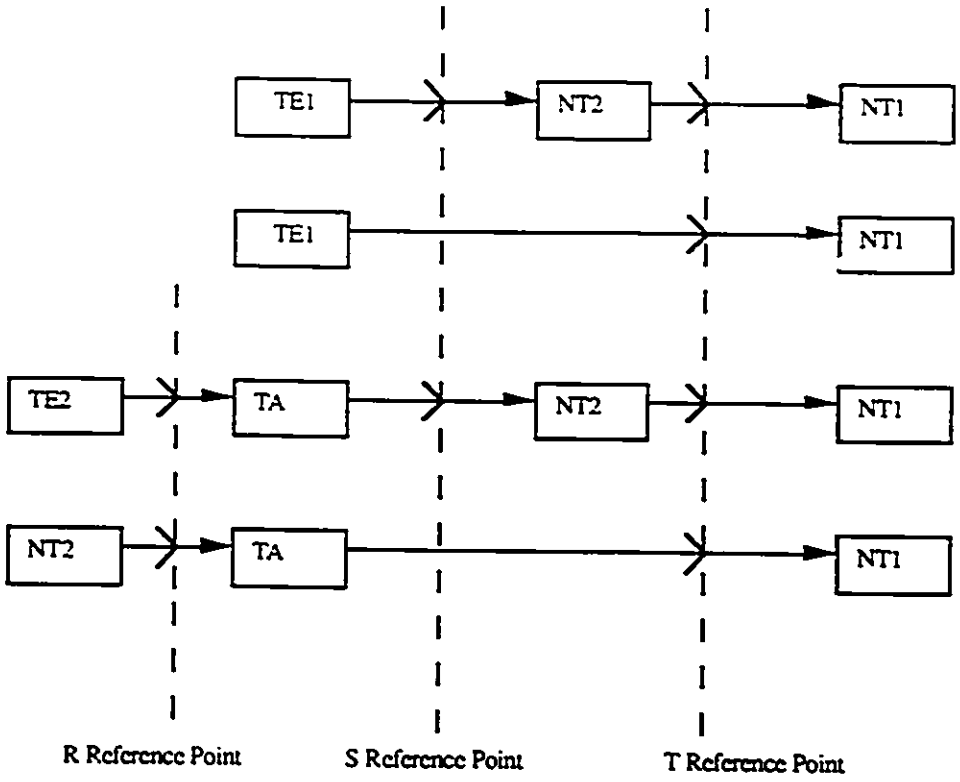


Figure 5.1 Standard ISDN Interface Reference Points

5.6 Physical Layer Configuration

Physical layer is the lower most layer of ISDN. This layer will directly operate on the hardware base. The hardware base has to comply with the ISDN recommendations I.430. The functions provided by the hardware across the interface are timing functions for synchronization with the remote, D-Channel access control, power feeding, and activation/deactivation, maintenance functions together with other characteristics such as frame structure and appropriate encoding of bit streams for both B and D channels. In order to have a greater appreciation of ISDN Q.931 protocol design, it is desirable to have a better understanding of the hardware operation which is given below.

Like any other communication device, ISDN will send and receive information from the peer through the networks in the form of streams of bits of Zeros and Ones, based on certain criteria of physical media of transmission and ISDN hardware support . The Ones and Zeros to be transmitted are grouped into a layer 1 frame of 48 bits. Thus information along with hardware synchronization bits will be transmitted as frames of 48 bits at a time. The frames are transmitted at a rate of 4000/sec. Therefore the total ISDN Basic Interface operates at 4000*48 = 192 Kbps.

Each frame consists of 8 bits for B1 channel and 8 bits for B2 channels and 4 bits for the D channel. The remaining bits of the frame are used for various other hardware functions such as synchronization between user equipment and network, collision detection for support of passive bus, frame alignment etc. It is not essential to go into further details of physical frames for understanding of this thesis.

It is the physical layer protocol which breaks down the layer 2 frames into the physical layer frames before transmitting them to the remote. It also buffers the incoming streams of bits, and constructs the layer 2 frames before passing it to the layer 2. In the case of erroneous transmission or loss of synchronization with the network or NT2, the hardware will make use of the control bits in the physical frame to recover from the errors and may purge the contents of the buffers.

5.7 Data Transmission Techniques

From layer 2 onwards the inter-layer protocol exchange is purely software oriented.It is important to picture how a frame, packet or a message is transmitted on the available channel bits through the hardware. According to OSI terminology, the layer 3 information exchange unit is called the packet, the layer 2 information exchange unit is called the layer 2 frame, the layer 1 information exchange unit is called the physical layer frame. According to ISDN terminology the layer 3 information exchange is called the *Message* while the layer 2 and physical layer information units are same as in OSI.

According to OSI reference model the layer 3 information exchange unit is embedded as the data to be transmitted to the peer by layer 2 information frame and the layer 2 information frame is embedded as the data to be transmitted by the physical layer frame. The information unit of each layer will consists of its own header and a tail along with the data to be sent.

For example, in the case of ISDN, let us consider a layer 3 message, namely CONNECT Message to be sent to remote. The above message has to be transmitted as the data in an I frame of layer 2. This I frame of layer 2 will be transmitted in the data frames of layer 1.

Layer 3 Message M		= Protocol Discriminator + Call Reference + Message Type + Channel Identification
		= 80018107180181
Layer 2 Frame	F2	= H2 (I2) T2
	I2	= (layer 2 data+M)
Layer 1 Frame	F1	= H1(I1)T1
	I1	= (Layer 1 Data+F2)

The above packet is a signalling packet and let's assume it is *n* bytes long. The entire frame which is *n* bytes long gets transmitted on the D channel (4 bits in a single frame for basic access). It will therefore need *n/4* frames for the entire signalling frame F1 to be transmitted. The remote will wait for at least *n/4* frames to be received before forming a layer 1 frame(assuming all good physical layer frames were received) in its ring buffer. The remote physical layer will then strip off the layer 1 header, tail and layer 1 data. If the contents of the header and tail and the layer 1 data are in accordance with the ISDN layer 1 protocol, it will then direct the I2 frame to layer 2. From now on it is complete software processing in the layer 2 and layer 3 protocols.

If the layer 1 frame is erroneous, layer 1 will behave according to its protocol for example, it may send a reject frame, until the right frame is received. It is only then that the physical layer will pass F2 to its layer 2. But depending upon the local layer 1 implementation, for erroneous frames it may log the errors or pass certain primitives to the higher layer or special management layer for human interference or higher layer decision.

The above behavior repeats at data link layer also. If the layer 2 frame is found acceptable according to LAPD data link protocol, the peer-to-peer layer 3 Message (Information part of the layer 2 frame) will be passed on to the ISDN Q.931 (layer 3) for peer-to-peer analysis. The peer-to-peer interactions of ISDN Q.931 are provided in chapter VI.

Similar data transmission techniques are applicable to the transmission of data on the ISDN B channels. The data exchange on the B channel itself may be through certain data exchange protocols modelled in accordance with the OSI layer concept.

5.8 Network Handling of the Protocol Data Units

The network itself may split the user Message and package it according to its own requirements and may break or combine multiple packets and may transmit the network data units on different virtual circuits or the same virtual circuits on the same physical channel or different physical channels. This thesis will not cover the network aspects of data transmission.

A general literature of interest on private and public networks for both OSI and Non-OSI architectures is given in [NET81].

In this chapter we provided an overview of ISDN concepts which will help in the design of ISDN Q.931 peer to peer protocol in the next chapter.

Chapter 6

6. Protocol Specific Design Concepts for the Example

To recapitulate, one of the aim of the thesis is to apply our generic protocol design concepts to a real world protocol. The chosen protocol is ISDN Q.931. We follow the major steps suggested in section 3.2 in the design of implementation-directed specification of ISDN Q.931.

It was necessary to study the CCITT ISDN standards namely,I.430, I.440, I.441, I.450 and I.451 in order to understand the ISDN Call Control Layer or ISDN Network layer (Q.931) and its existence in the global environment. Section 6.1 in this chapter has resulted from the study of the above specifications.

It is important to recall that Q.931 is only a signalling protocol, which interacts with the network and remote on the separate signalling D channel to negotiate for the bearer capabilities of the local ISDN terminal.

The bearer capability negotiation includes the negotiation of terminal features and capabilities such as the following:

Information Transfer Capability
Transfer Mode
Information Transfer Rate
Structure and Configuration
Symmetry of Protocol
Layer and Protocol Identification

Exact details and negotiation parameters can be found from the Bearer Capability Information Element description in the CCITT I.451 standards.

6.1 Signalling in ISDN

Now that we know ISDN Q.931 is purely a signalling protocol, let us evaluate the components of signalling.

The signalling takes place through exchange of Q.931 protocol units called the *Messages*. There are a finite set of messages meant to accommodate *negotiation*, *sustenance* and *release* of an ISDN call during every possible situation arising during the course of an ISDN call.

In order to keep the set of ISDN messages a minimum, and yet be able to communicate larger signalling information, the messages are composed of smaller information units called the *Information Elements*. The messages and the Information Elements have a specific syntax.

The immediate task now is to get a state machine for exchanging of the messages. Due to the presence of the sub-elements of the protocol data units, the transitions of the state machine is now dependant not only on the Message but also on the Information Elements present in it.

As part of the Step 1 of the design process as explained in section 3.3 we have arrived at the following major factors which are essential for the Step 2 of the design process.

There is the concept of Mandatory and Optional Information Elements in Q.931. The Mandatory Information Elements should be compulsorily present in a message for the message to successfully participate in the semantics of Q.931 behavior. Likewise, it is optional for certain Information Elements to be present [Tutor90]. As a first step in the study of the specification, classify Information Elements of all the Messages as Mandatory, Optional or find out if they have any other attribute associated with them. From the Q.931 specifications and other literature the following assumption are deducible:

1. A simple ISDN terminal need not support all the Messages
2. Even if all the messages are implemented the network itself may not support all the features.
3. The support of every additional feature will cost additional money to the customer.
4. Even if a Message is supported, not all the Information Elements are essential.
5. An implementation must be able to receive any message with any Information Element in it, even though it may not send it. One way to handle such an IE is to ignore it.

Thus it is very important to understand every sentence of the specification before its implementation commences. Efforts must be made to gather information from other implementors especially the network vendors to find out about the features they will be

supporting etc. After gathering all the above information, a basic state machine can be designed with a basic set of information elements to support negotiation of a minimum set of terminal bearer capabilities.

The minimum and valid signalling required to establish a voice or data ISDN call is shown in ISDN signalling flow diagram in figure 6.1. The intention of figure 6.1 is to make the concepts of ISDN Q.931 clearer. Once the concepts of ISDN signalling are clear (so far it has accounted to step 1 of section 3.2), state machine for ISDN Signalling can be generated as shown in figures 6.2 and 6.3, which are obtained from [CCIT3] [Tutor90] [UyLa90]. Based upon the understanding of the ISDN signalling flow diagram and the state machines, a discussion on the design of the user primitives for the ISDN signalling is presented in section 6.2.

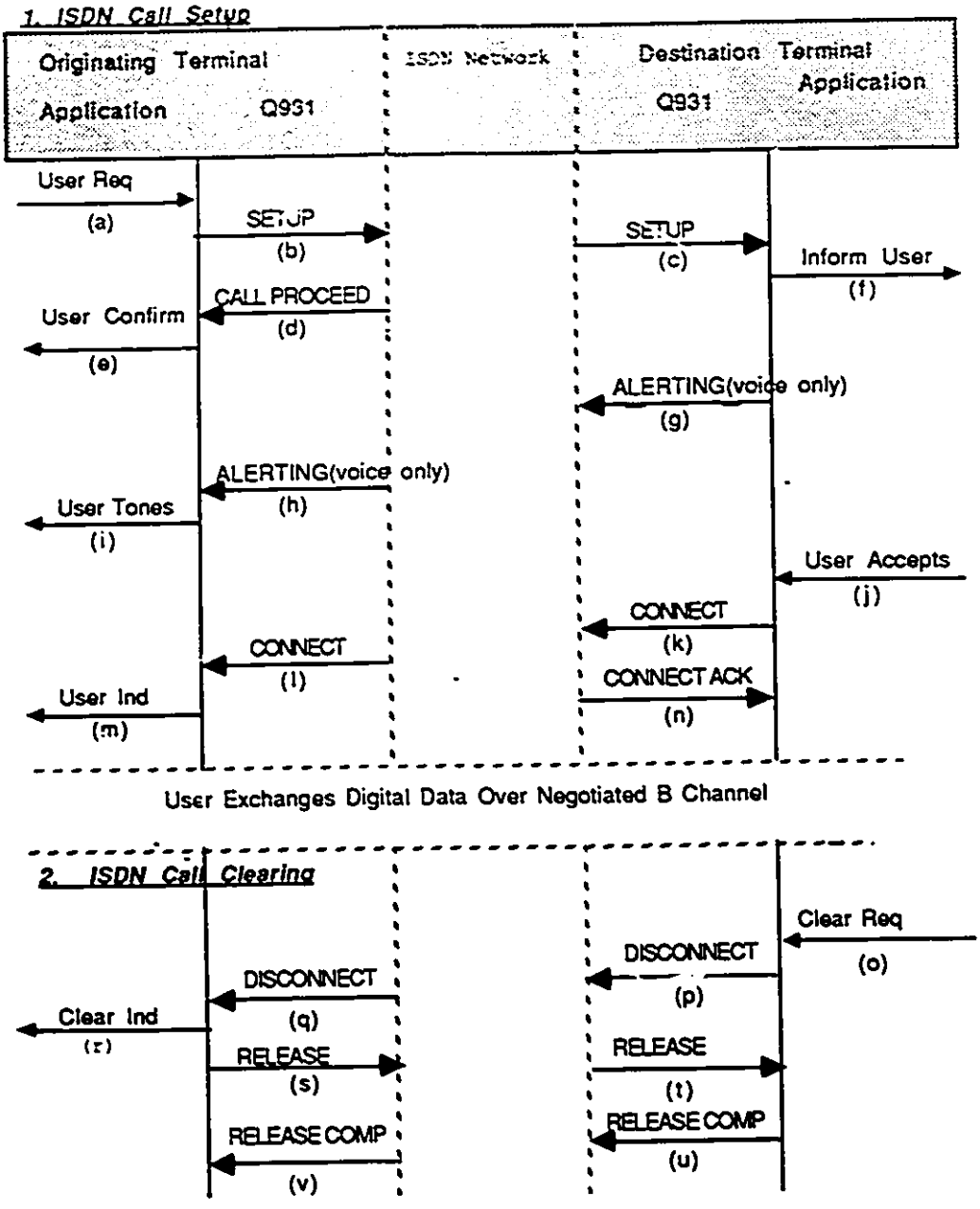


Figure 6.1 ISDN Signalling Flow Diagram

Let us try to understand the significance of the above signalling:

I ISDN Call Establishment or Setup

1. In the above diagram the user of the origination equipment decides to talk to another user. He does so by invoking an application software(a). The user should provide parameters such as destination address, the type of information, the mode of transmission, the type of data channel etc. This application will invoke the ISDN Q.931 and will provide the parameters for an ISDN call to be setup.

2. The local implementation of Q.931 will format the SETUP message with all the required Information Elements based on the user values, and its own intelligence. SETUP (b) is the first message which should be sent in order to set up an ISDN call.

3. The network will send the received SETUP message to the destination(c) and at the same time inform the origination side that the call is proceeding to the destination by sending a CALL_PROCEEDING message (d). The local Q.931 may at this time send a confirm to the user (e) or wait until it gets either (h) or (l) before this confirm is sent together with the resulting indication (either i or m). It is a local design issue.

4. The destination Q.931 will now inform its user the reception of a valid SETUP message through an User Indication (f). If it is a voice call then the destination user's ISDN bell will ring and the destination Q.931 will send an ALERTING message (g) to the origination side.

5. The reception of the ALERTING message (h) on the local may trigger an indication to user as in (i).

6. The destination user's action (j) (for example picking up of the telephone in the case of a voice call) results in the destination Q.931 sending a CONNECT message (k).

7. On reception of (k) the network will send a CONNECT ACKNOWLEDGE (n) to the remote user and CONNECT(l) to the local user. On reception of CONNECT (l),

the local Q.931 will send a local indication (m) to the local user suggesting that the negotiated call is now successfully connected.

The ISDN call is now said to be established. At this stage the B channel has been completely negotiated and the Data transfer can now take place on the B channel.

II ISDN Call Clearing:

1. Let's assume that the remote user decides to take down the call by hanging the voice set(o). This prompts the remote Q.931 to send a DISCONNECT (p) message to the network. At this time the B channel which was negotiated above will become free for another call.

2. The network will send a RELEASE message (t) to the remote Q.931. At the same time the network will send the DISCONNECT(q) message to the local side.

3. The remote Q.931 will send a RELEASE COMPLETE(u) message to the network suggesting that the call is completely released and the call reference may be used for another call.

4. On reception of the DISCONNECT message (q) the local Q.931 will inform the user by sending a indication(r) suggesting that the call is being cleared. At this time the B channel becomes free.

5. The reception of the DISCONNECT will prompt the local Q.931 to send a RELEASE message (s) to the network. As a result the network will send a RELEASE COMPLETE message (v) to the local Q.931. The call reference which was used for the above call will now become free on the local side.

6.2 Peer-to-peer ISDN Signalling State Machines

This section accounts for the step 2 of Protocol Design Process. As was mentioned earlier, the CCITT specification provides SDL diagrams for peer-to-peer interactions.

The state machines of Q.931 protocol control procedures provided in figures 6.2 to 6.3 depicts all the major states required for an ISDN voice or data call to be established, however it doesn't include few optional states for Overlap Sending, Restart handling etc., The implementation of the above state machine will qualify the ISDN Network layer to be

operational. For a barebone ISDN terminal to be able to make use of ISDN services through an ISDN network, only a subset of I.451 messages and Information Elements needs to be implemented. In fact, at the beginning even the networks are not providing all the possible envisioned services. With the progress of time as ISDN becomes popular, we can see more and more of these services being offered by the networks.

The ISDN signalling can be represented by two state machine diagrams. The state machine diagrams were derived from the CCITT SDL diagrams and the accompanying explanation on message handling in the CCITT ISDN specification I.451. The state machine diagram in figure 6.1 represents the call connection or the channel negotiation procedures for ISDN Q.931. The state machine diagram of figure 6.2 represents the call disconnection procedures of ISDN Q.931. The Call Connecting and Call Disconnecting state machines were separated in order to preserve legibility in the diagrams. The names on top of state machine transitions are the events. The peer-to-peer and a few inter-layer events are mixed up. The inter-layer events have the prefix NL attached to the name. The + sign preceding the event indicates it is an incoming event while the - sign indicates that it is an outgoing event. There may be multiple outgoing events, for example one to user layer and the other to the peer entity. Only those inter-layer events are shown which are indicated in the SDL diagrams of the CCITT I.451 specification. However, for the state machine to satisfy the design steps of section 3.2 we should add more primitives to the above state machine. The need for additional primitives is discussed in the next section.

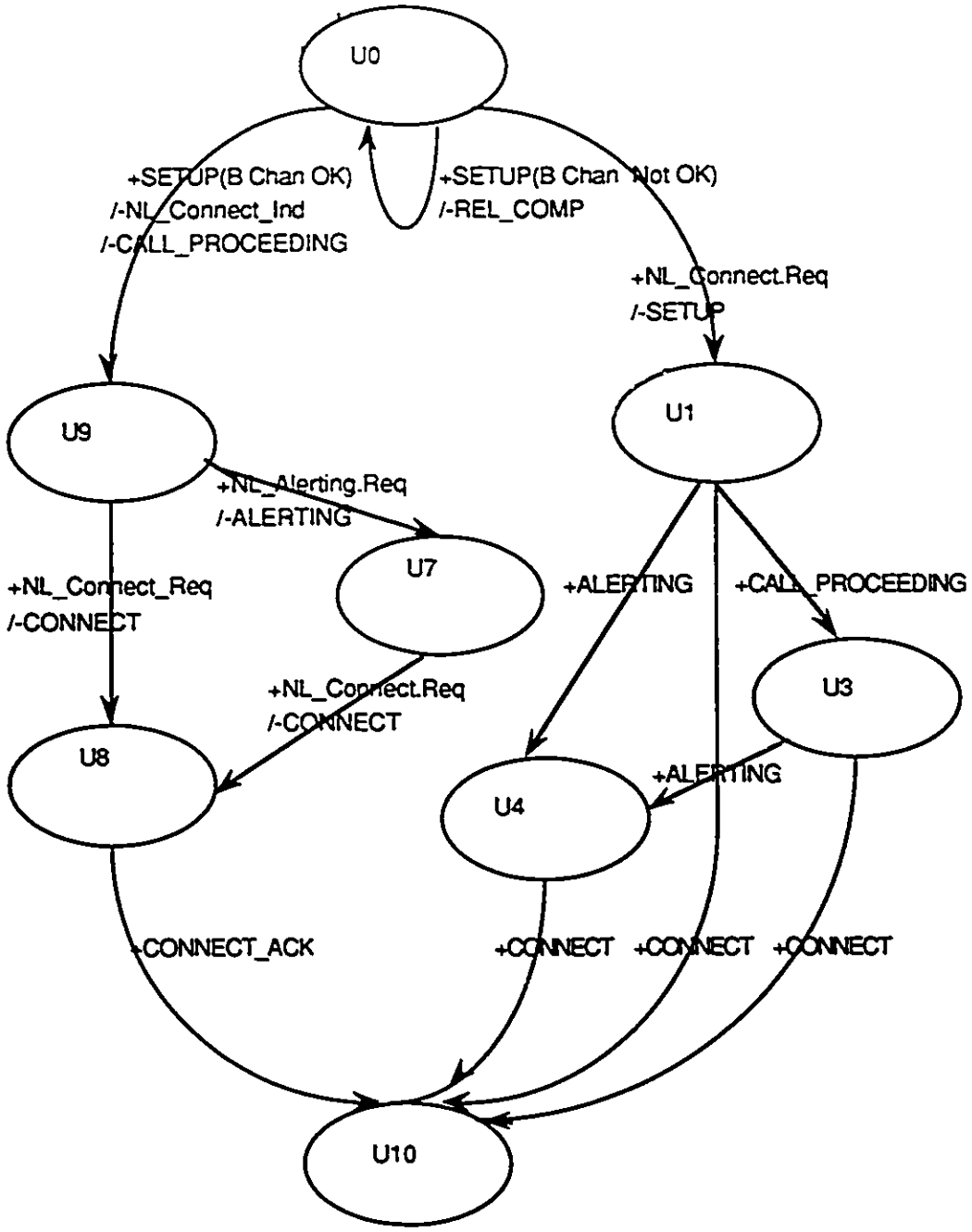


Figure 6.2 B Channel Negotiating Procedures For Q931

B Channel Disconnection Procedures of Q931

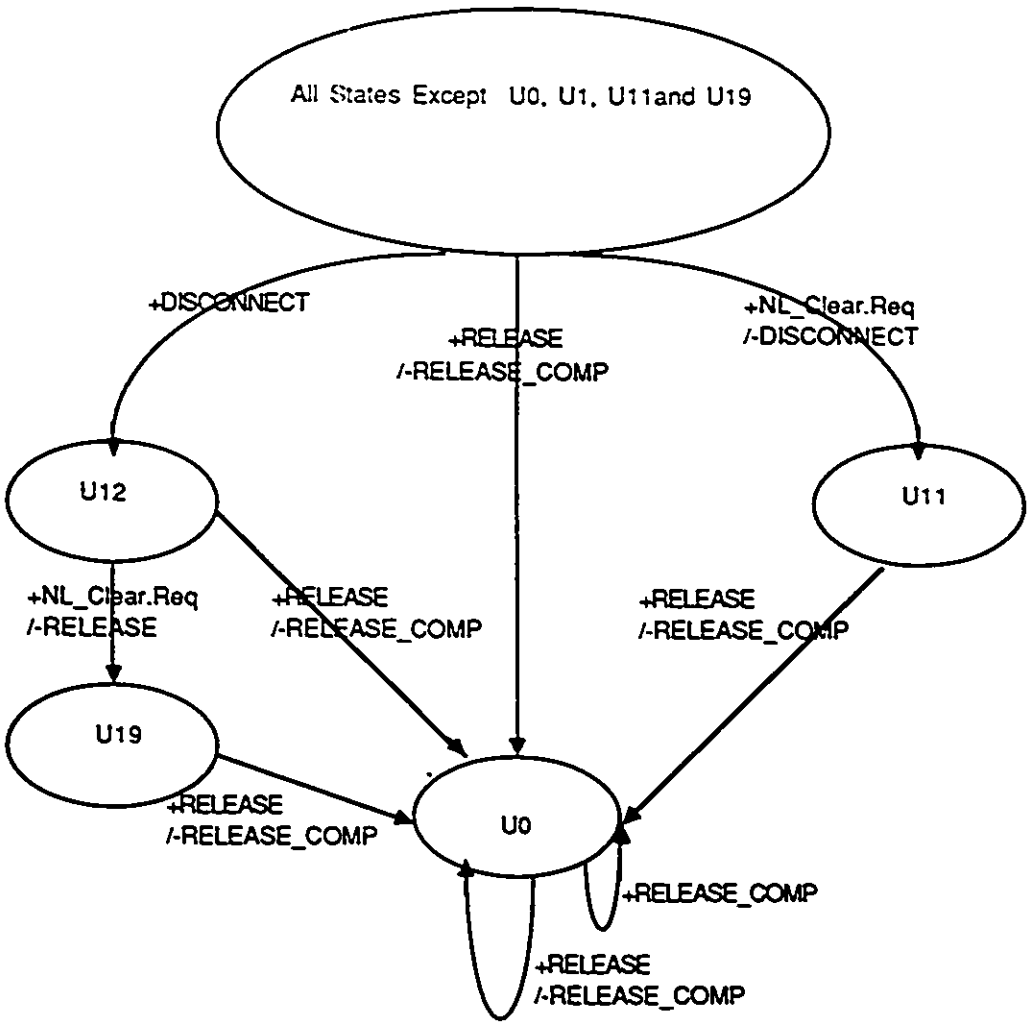


Figure 6.3 B Channel Release Procedures

6.3 Design of sample user primitives to support ISDN signalling

This section is indicative of the design of user primitives based upon the protocol service requirement initially discussed in section 3.8.1 and later in section 3.8.1.3.

Design of User Primitives based upon the Services provided by the Protocol under development

In the above diagram the user request(a) was desirable in order to enable Q.931 to format SETUP message before being despatched to remote. In the request (a) the user should provide all the informations which are required for the SETUP message to be formatted. It is a design issue as to what parameters the user will send, the other alternative would be that the intelligence to format the above parameters may be provided in the Q.931 itself. Factors such as usability, versatility, performance of code etc, are to be considered while designing parameters for (a). There are always some parameters such as address of the remote user which have to be provided anyway by the user setup request(a). In the CCITT SDL diagram this request is called *Connect Request* and in figure 6.2 it is called NL_Connect_Req. Since NL_Connect_Req is specifically meant to generate the SETUP message, we will call it as NL_Setup_Req in the Estelle Specification given in Appendices.

Like wise the reception of SETUP message by the remote should be informed to the user through an indication (f). Unfortunately, the CCITT ISDN I.451 specification doesn't indicate this primitive in the provided SDL diagram. However, the author has read the Specification carefully, and as a part of the design process, it was evaluated that the incoming SETUP does bring certain important peer end-terminal characteristics and therefore those characteristics should be notified to the higher user of the Q.931. In figure 6.1 the indication (f), should deliver user information such as address of the caller, the type of call, and call related information. Depending upon this information, the user will either accept the call or reject it through a response (not shown in the diagram). Again it is a design issue, the entire intelligence may be provided in the Q.931 itself so that it may reject the call directly or else the appropriate information is sent up like in this case and the user has to decide about the call.

In (a) the user had requested certain parameters to be sent out such as the selected data channel. The network may either accept it or choose an alternative as pointed out in message (d) also in the event that the message (b) was lost the requester should be informed through the confirmation (e).

According to ISDN Q.931, the ALERT message can bring certain audible tones and other signalling user data with it, which will be sent to the user through an indication (i). Based upon the information the call can be accepted or disconnected by the user through a response (not shown in the figure).

Once the call is connected, the user can send data through the B channel. The completion of the call connection is informed to the user through an indication (m).

In order for the user to start disconnecting a call he will be provided with the primitive (o). The beginning of the *releasing* of a call (reception of DISCONNECT) will be indicated to the user through an indication (r) and another primitive may be provided which will inform the user about the completion of the call clearing.

In general, whenever there is some signalling information which should be conveyed to the user in order for him to take appropriate action (For eg., the user may decide whether to accept or reject a call based on the B channel selected by the network (remote). A second example is that, the user may keep record of signalling information (for eg., DISCONNECT and RELEASE COMPLETE messages may bring with them the information related to the duration of the call such as charges for that duration), in such cases the user primitive should be provided.

The general location of user primitives to access the peer-to-peer ISDN call control features of Q.931 are shown in figure 6.4. The user interactions to access few of the Q.931 features are shown in figure 6.5.

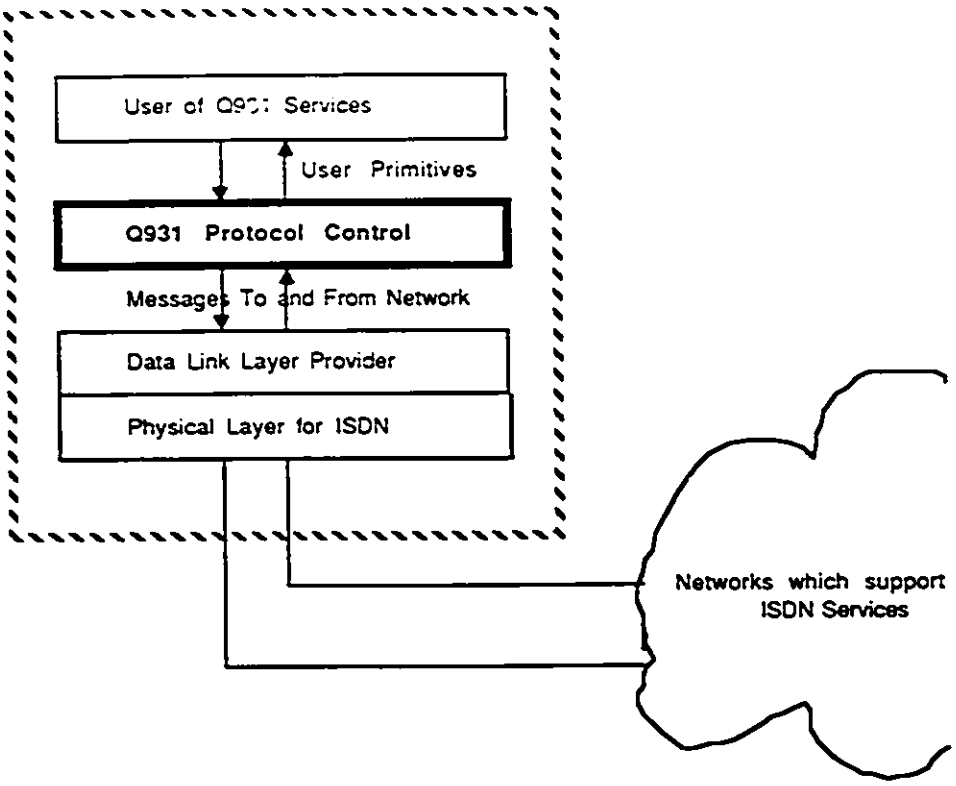


Figure 6.4 Position Of Q931 Protocol Control Layer In A Global Isdn Picture

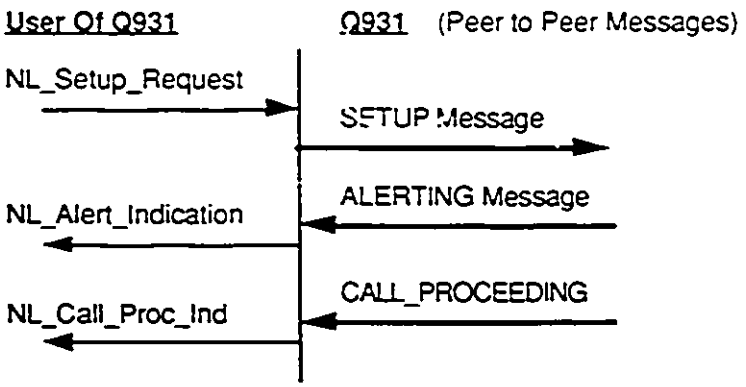
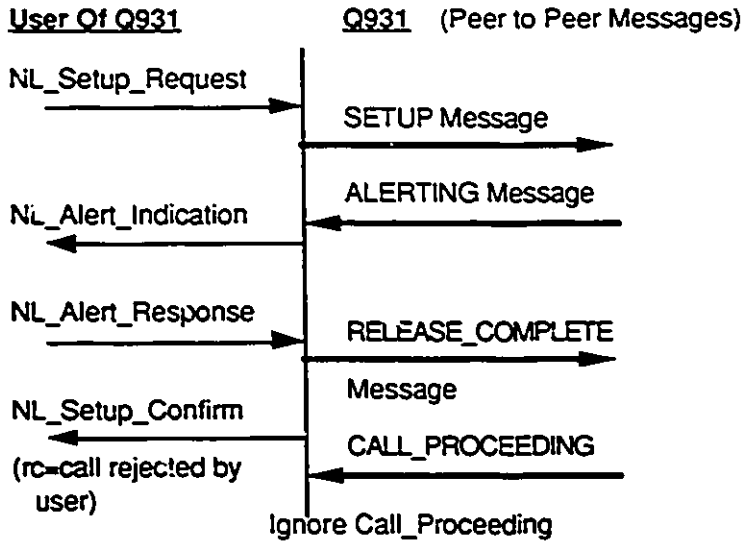


Figure User Interaction to show a valid setup of call



User Interaction to show rejection of a call based on Alerting Tones

Figure 6.5 Sample ISDN User Interactions

Design of the User Primitives based upon the Services Provided by the entire underlying Protocol Stack

The issues discussed in point 2 of the section 3.8.1.3 hold true for the activation of service and deactivation of service of the underlying protocol stack. However, the

primitives which the protocol under development (Q.931) will issue as a result of the user primitive of section 3.8.1.3 will be different because of the different SAPI and TEI concept as discussed in next section.

6.4 Design of Primitives to make use of Underlying Service

The underlying protocol layer for ISDN Q.931 is ISDN LAPD. ISDN LAPD is a data link layer protocol and it is similar to the data link layer LAPB procedures. A few design issues pertaining to the design of interaction of LAPB with its user layer were discussed in section 3.8.1.2 and shown in figure 3.7. All the interactions of LAPB exists in the interactions between Q.931 and LAPD. A few additional interactions are also required, which will also explain the differences between the LAPD and LAPB protocols.

The first major difference between LAPB and LAPD protocol is, LAPD consists of two octets of address called the Connection End Point Identifier. The major contents of two address octets are the Service Access Point and the Terminal End Point Identifier.

The significance behind the two octets for the address of the LAPD frame is shown in figure 6.6 (adapted from CCITT ISDN I.440) and explained in the following paragraph. ISDN user premise equipment NT2 can support upto eight end user equipments (TE1's or TE2's). Each end user equipment can communicate with its peer and make use of all the services offered by ISDN. Therefore at any time multiple end user equipments can communicate with their peers through NT2. Since multiple end user equipments communicate to peers through a single NT2, the connections between the end user equipments and the NT2 is identified by the Terminal End Point Identifier field of the LAPD address octets.Therefore each end user equipment gets a unique TEI value.

Within each end user, the ISDN standards provide the ability for the user to have multiple entities. The multiple entities could be the ISDN Q.931 protocol itself, the management services of ISDN Q.931, the X.25 data communication protocol etc., Therefore each entity gets a unique Service Access Point Identifier. The SAP Identifier for Q.931 protocol is 0.

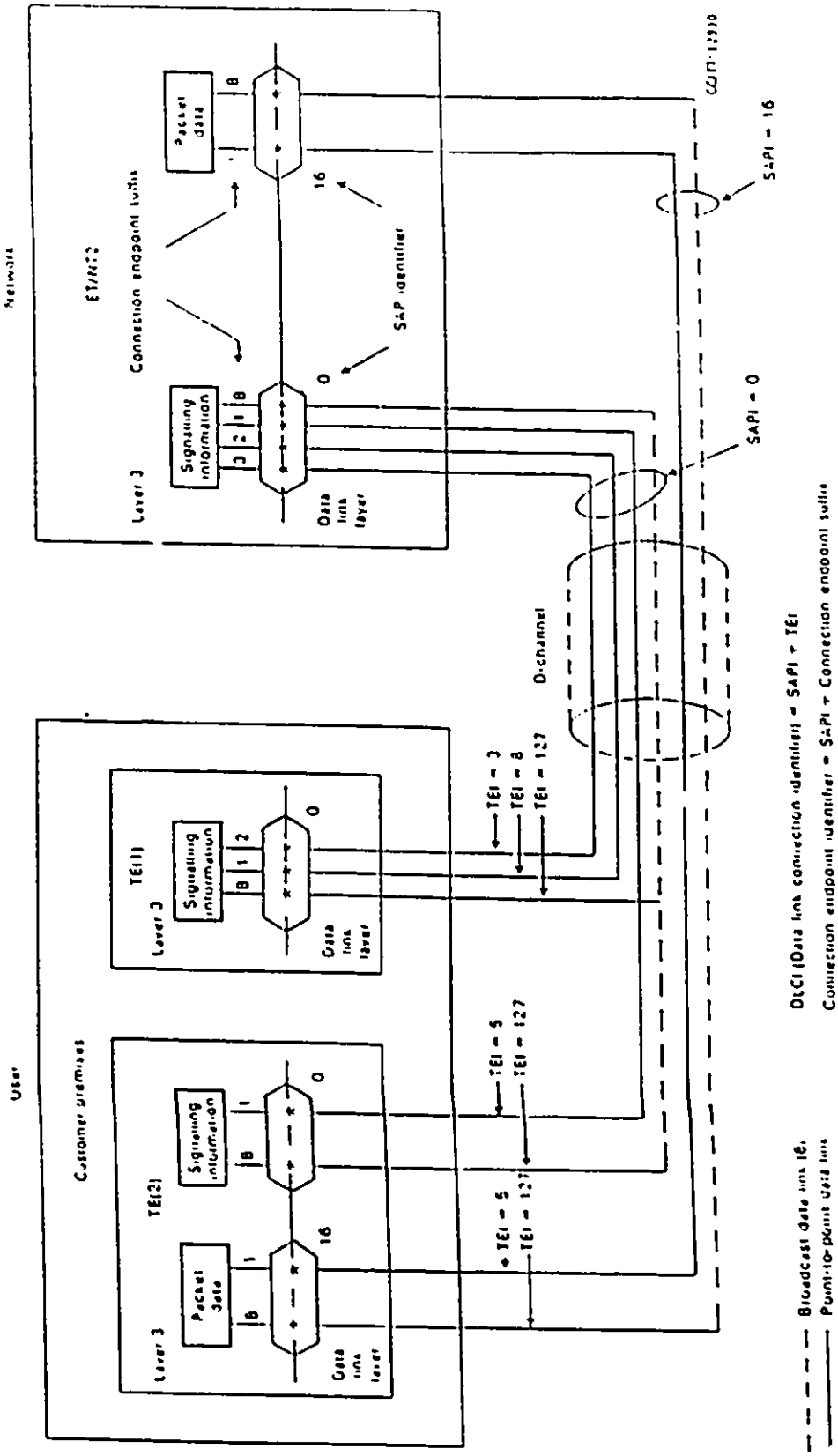
It is also possible that a single end user equipment may have multiple LAPD's.

The Higher Layer Service Activation as shown in figure 3.8 will be changed to the figure 6.7 to accommodate the concept of SAPI and TEI in LAPD.

In this chapter we have presented in detail the design issues in the generation of service primitives which map into peer interactions. These service primitives are then merged with the service specifications developed in chapter 3 and 4. In the next chapter a subset of generic ISDN Q.931 specifications will be specified using Estelle to which the

service specifications developed in this chapter will be added, resulting in an implementation-directed Estelle specification.

Figure 6.6 Relationship between SAPI, TEI and data link connection endpoint identifier.



Note – The management entity is not shown in this figure

FIGURE 3/Q.920

Overview description of the relation between SAPI, TEI and data link connection endpoint identifier

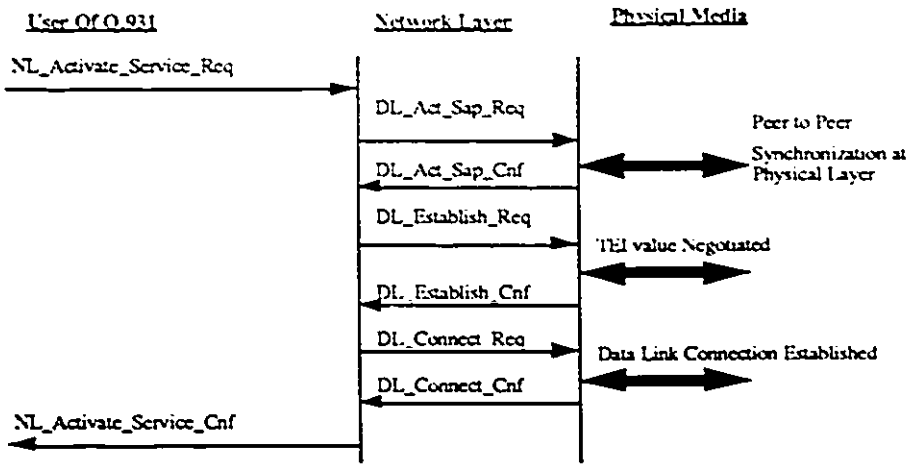


Figure 6.7 Merging Of Primitives for Q.931

Chapter 7

7. Implementation-Directed Formal Specification of ISDN Q.931

7.1 Introduction To Estelle

Estelle is a formal description technique adopted by ISO [ISO3] to serve as a means to remove ambiguities from ISO protocol standards. It is possible to specify distributed software and hardware functions using Estelle. Estelle is an extension of earlier work done by [Ansa82], [Boch78] and [TeBl81]. Estelle is a set of extensions to ISO Pascal [Pascal1] which allows the components of a data communications protocol to be modelled.The introduction to Estelle along with its syntax and semantics is in [BuDe87] and the features of Estelle useful for specifying distributed systems is in [Linn86].

The distributed system in Estelle is a collection of one or more communicating components called the *module* instances.Each module instance will have certain inputs and outputs. The exchange of inputs and outputs will be through the *interaction points*.An Estelle Specification will have certain *Static Modules* and may contain *Dynamic Modules*. Therefore, any Estelle Specification is a *hierarchy* of modules. The *static* modules will have a pre-defined hierarchy while the hierarchy of dynamic modules will change with global behavior of the system. The presence of hierarchy of modules is also called *nesting* of modules.

There are two kinds of interaction points,namely, *internal and external* interaction points. An *external interaction* point is global to the specification system, which means different modules of the specification can communicate through them after appropriate dynamic or static linking. An *internal interaction* can exist between a parent module and its child module.

A module may contain a state automaton which is non-deterministic. A module is said to be *active* if it contains at least one transition, else it is called *inactive*.

In an Estelle specification, the hierarchy of modules consists of a main module called the *system module*. The system module will have the attribute *systemprocess* if it consists of modules of attribute *process* and/or *activity* within its hierarchy. The modules within the hierarchy of *systemprocess* specification can execute in parallel. Similarly, the system module has the attribute *systemactivity* if it consists of modules attributed only *activity* within its hierarchy. The modules within the hierarchy of *systemactivity* specification cannot execute in parallel.

A complete Estelle specification hierarchy of modules including the main system module and all the nested modules is called the *sub-system*.

If a module invokes another module from within its body, a relationship of parent and child is said to exist. This relationship can be represented by the edges of a tree.

The following rules should be followed in *attributing* a module during nesting:

1. Every *active* module should be attributed.
2. The system module cannot be nested within any attributed module.
3. Modules attributed *process* or *systemprocess* may contain nested modules attributed either *process* or *activity*.
4. Modules attributed *activity* or *systemactivity* may contain modules attributed *activity* only.
5. A module containing the system module should be *inactive*.

Once the modular partitioning of the distributed specification is decided and the *interactions* through the *interaction points* are decided, the interaction points should be linked. When an external interaction point of a task is bound to an internal interaction point of its parent task, such interactions are said to be *attached*.

Similarly, two interaction points are said to be *connected* if:

1. both are external interaction points of two sibling modules.
2. one is an internal interaction point of a module and the other an external interaction point of its parent module.
3. both are internal interaction points of the same module.

At any time, only one interaction point of a module should be connected to atmost one other interaction point. An internal interaction point of a module may at any time be attached to atmost one external point of the parent module and atmost one external interaction point of its children modules. There cannot be a bi-directional connection in an *attached* interaction point simultaneously.

Communication Mechanism

Two types of communication mechanisms are used in Estelle

1. message exchange through Interaction Points and
2. restricted sharing of variables

Message Exchange through Interaction Points

Communication can take place between two modules through a previously established interaction point. A message which arrives at the external interaction point of a module is put in a FIFO queue. There is no limit on the number of interactions which can be put in this queue, hence the queue is said to be *unbounded*.

The unbounded FIFO queue belongs to only one particular interaction point of a module, it is called the *individual queue*. If the FIFO queue is shared by a number of modules it is then called the *common queue*.

A source module can always send a message irrespective of whether the target module is ready to receive it or not, this type of communication provided by Estelle is called the *non-blocking* communication as opposed to *blocking* or *rende-vous* communications.

Modules which are linked directly through their interaction points are said to have a direct connection while the modules which are indirectly linked through several attachments are said to be indirectly connected.

Message Exchange through shared variables

Certain variables can be shared between parent and child modules. Shared variables should be declared as *Exported* variables in the child process. A parent module always has a priority over the child modules and hence the problem of simultaneous updating of the shared variables is eliminated.

Parallelism & Non-Determinism

Two kinds of parallelism behavior exists in Estelle modules:

1. Asynchronous Parallelism and
2. Synchronous Parallelism

Asynchronous Parallelism exists between modules belonging to different sub-systems, while the Synchronous Parallelism exists between modules belonging to the same sub-system. According to Estelle semantics, multiple modules of the systemprocess sub-system which are attributed can execute in parallel at once, while only one module from the systemactivity sub-system can execute at any time.

Also a declaration part of a module within the sub-system main module may nest other module definitions which may in turn include other module definitions, and so on. Thus the way the existing module instances of a specification will behave with respect to each other depends on the attributes of the nested modules, and the parent/child priority.

Global Behaviour of Estelle Specification

The global behavior of a Distributed System specified in Estelle is defined by the set of all the possible sequences of *global situations*. After the execution of every transition, there may be a difference in any two global situations.

In Estelle it is not possible to represent intermediate global situations between one transition. Therefore Estelle transitions express *atomicity* in their execution.

The operational semantics for Estelle describe the way the transitions may be interleaved so that the specifications will have synchronous parallelism within the subsystems while maintaining an asynchronous parallelism between the subsystems.

7.2 Specification of Simplified ISDN Q.931 using Estelle

This section will first highlight the important features of Estelle and will depict how the Q.931 behavior can be specified using the applicable Estelle features. Discussion of Estelle features is limited to Q.931 applicability.

As part of experience of developing the specification of simplified ISDN Q.931, the author has acquired certain techniques to model specific features of a protocol in Estelle. These techniques will be discussed in this section. This section will also perform a critical analysis of Estelle features whenever certain Q.931 service requirements cannot be suitably specified or if the Estelle specification is not very elegant. The author will propose certain set notation extentions to Estelle transitions to reduce the number of Estelle lines required to specify a particular transition. These extensions can be enhanced and adopted within a local environment.

Nesting of OSI Layers in Estelle

OSI layers are normally represented as vertical stack as shown in figure 2.1. This familiar vertical stack is represented as horizontal modules in Estelle. The vertical relationship in OSI layers is exercised through the Service Access Point(SAP) by the exchange of inter-layer primitives(interactions). In Estelle the vertical relationship(father and son) in the hierarchy of modules doesn't represent the above OSI interaction or SAP, instead it means that all the interactions which arrive at the interactions of the father will also be seen by the son if the modules are *attached*. The vertical relationship is indicated by

the *attached* linking of Estelle IPs. The vertical layered relationship of OSI is represented in Estelle by having the Estelle modules in a horizontal relationship and the linking between IPs of modules is through *connect* feature of Estelle. From the experience of the author, the above distinction is not clearly explained in the Estelle literature and therefore the concept of *connect* and *attach* are very misleading. The static hierarchic model of the entire ISDN specification which consists of specifications for User of ISDN Q.931, ISDN Q.931, and ISDN LAPD merged with ISDN Physical Layer is shown in figure 7.1.

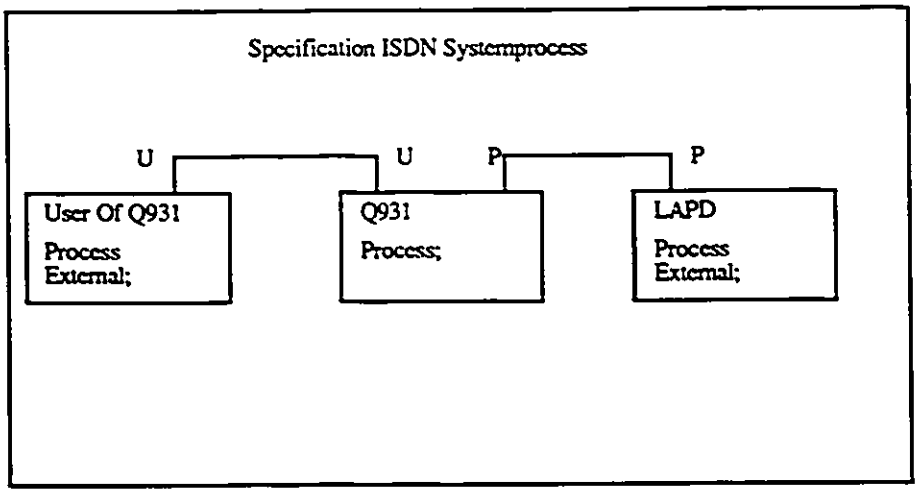


Figure 7.1 Static Hierarchic Structure And Interaction Links of ISDN Specification

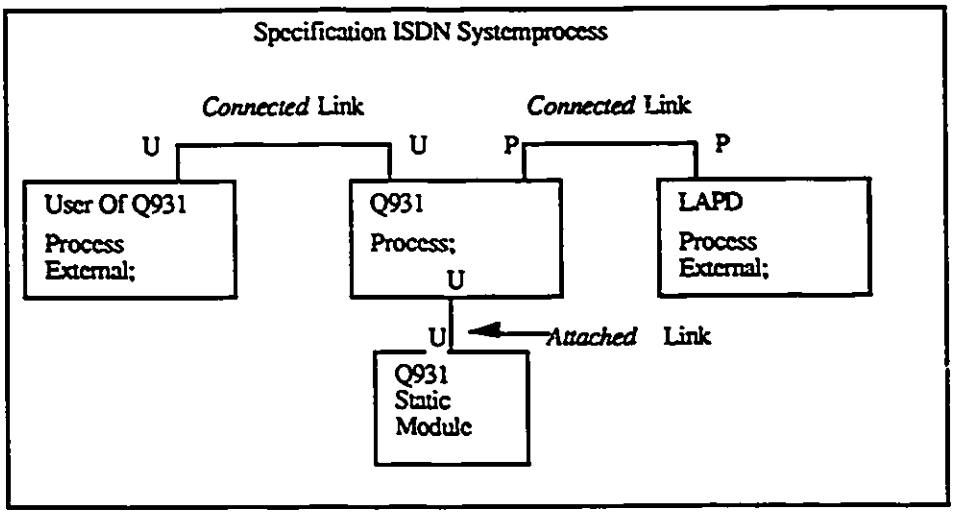


Figure 7.2 Static Location Of Q931 Module In ISDN Specification Hierarchy

In the specification of protocol of any OSI layer, the adjacent layers will play very important roles. In this thesis only ISDN Q.931 is specified, the adjacent layers will be treated as *external*. The external key word means that the specifications for the layer is elsewhere.

The entire ISDN specification will be treated as a *system* module. The attribute of this main system module will be *systemprocess*, because it is possible for each layer of ISDN specification to execute asynchronously parallel. The main system module itself is not nested within any module. The specification of layers adjacent to ISDN Q.931 will be treated as different Estelle modules which will have the Estelle qualifier *external* following the Estelle module type definition. The attributes of the User of Q.931, Q.931 and LAPD modules will be *process*. The layers of ISDN are specified as modules in order to accommodate the OSI multilayer functional partitioning. In fact real protocol layers are often implemented as different physical tasks (process) each with well defined interactions points and service primitives. The Estelle *module* feature is therefore very useful for specifying the functional behavior of different OSI architecture layers. This feature is also useful for specifying any real distributed system [Phalip89].

The above Q.931 adjacent layer modules are initialized through the execution of Estelle *init* clause at the end of the ISDN specification, and the modules are linked using the *connect* which permits hierarchic relations as defined by OSI architecture. The *attach* clause is not made use of in the ISDN specification at this time. The static module interconnection structure is also shown in fig 7.1.

Estelle Specification for a protocol which runs on a Uni-Processor System

The author has made the following observation. Estelle has the ability to specify a real distributed system protocol (which runs on different CPUs) by splitting up the realtime execution into different modules. This feature permits specifications of a single processor based multiple threaded (different processes) protocols as well. The creation of multiple processes is desirable in order to model the protocols into separate functional groups as specified in the standards. Sometimes the creation of multiples processes becomes a necessity in order to permit the time sharing of the system resources.

Normally, the *network stack* itself is implemented within a sub-system which runs on a single processor. However, it is possible that a single layer protocol entity may be implemented as two virtual tasks (processes) such as a task for the receiver state machine and the other for the sender state machine. Every layer in itself is separate; in other words each layer may have its own send and receive task.

The modularization of the Estelle specification is desirable for implementation purposes, but the generic specification which comes from the standards bodies should contain a minimum number of Estelle modules. A generic specification with excessive modularization will tend to force a design direction, thus leading to an over specification. However, an implementational specification should contain the necessary modules for the generation of the executable protocol code.

The level of modularization is a source of tension between the specification body and the development community. The generic specification should not overspecify, instead, let the development community introduce their own modularization in order to generate the executable protocol code from their implementation oriented specification.

The presence of excessive Estelle modules either in generic or local implementational specifications will make it difficult for synchronization between the modules. It will require detailed analysis based on the Global Situation Graphs of the Estelle Specification. A much better method of synchronization is claimed to be the Rendez-Vous mechanism (called synchronization clauses) [Cour87] [Cour88]. Hopefully it will become a part of Estelle standard. This mechanism will eliminate inherent priority and delay clauses within Estelle modules by synchronization through explicit semaphore exchange between modules.

In this thesis the Q.931 specification will use the existing Estelle standards(DIS 9074). The Q.931 specification being developed is assumed to run on a Uni-processor system.

Factors in nesting of any single protocol specification

Nesting of a protocol specification can be static and dynamic. Even though it is difficult to lay down specific guide lines for the nesting of modules, the author believes the following factors will assist in deciding. A generic specification should have minimal module nesting, because, there is an implementational choice in deciding it.

Static Nesting

-If a protocol implementation is partitioned into a receive machine and a send machine, the two state machines may be specified as two static modules.

-If a protocol layer serves a number of users, each user will have an independent virtual server state machine. Therefore the protocol layer may be partitioned into a number of modules [Boch88].

-If a number of users access the same hardware or physical layer module then they may all be statically *connected* to the same physical layer module.

-In general if a certain function in the specification can be isolated to execute independently, it may be represented as an independent static module.

Dynamic Nesting

-If the reception of a transition results in a set of complex operations such as multiple queries of a data base[DiDu89] then a dynamic child module may be created. It is especially desirable in order to maintain the *atomicity* of transitions desired by Estelle specifications.

-If the reception of an interaction signals a drastic change of behavior (for e.g., the reception of an interrupt which results in an exception handling) of the module, the handling of such receptions will make the transition automaton very complex. Such interactions should therefore be handled by a special dynamically created module. There will be as many modules as the number of such interactions.

Implementational Nesting of a Multiprocessor Distributed System

In specifying a system (not a single layer protocol) which runs on multiple physical distributed systems, parallelism is very important and Estelle allows such parallelism to be represented adequately. It will be useful to follow a three level hierarchical approach as shown below in nesting complex distributed or communicating systems.

(1)At the first level, there is parallelism between different physical units such as Systems, Printers, Networking interface etc.,

(2)At the second level, the parallelism within each unit of first level should be considered. For eg., the Networking unit may have simultaneous session at any time as a result of multiple networking cards within the system.

(3)At the third level each unit of second level may itself have parallelism, for example the normal and abnormal interrupt handlers.

Style of Estelle Specification

It is highly debatable whether Estelle specifications using detailed Pascal data structures is desirable or not. From the literature there are mixed opinions about the resulting over specification as a result of extensive use of pascal data structures.

The author's stance is following:

- o If the specification comes from the Standards Working Committees then it should be as generic as possible. Since most of the present day protocols are not specified using Estelle, the Estelle specification should be created from the formal English Specification by the protocol implementors themselves.
- o It is well known software engineering technique to write a high level design and refine it with appropriate algorithms and suitable data structures. The high level

design and detailed design is normally done using a design language particular to an organization or by simply writing pseudo code using conventional languages. The author encourages to make use of Estelle at this stage of protocol engineering process. As a result it is possible to have a design which is oriented towards communicating systems and it provides all the benefits of a pseudo code along with the benefits which result from the usage of formal description techniques mentioned in section 2.5.2.

- The Estelle specification is generated by the designers and developers of protocols which reveal the implementational details of a protocol down to algorithms and data structures used.The Estelle specifications will be produced in increasing versions indicative of the increasing details (similar to higher level and detailed design iterations). Since the Estelle specification is produced by the developers, the author suggests the production of what are called *Functional* and *Implementational level specification* in Estelle. It is suggested in [DiDu89] to keep two levels of Estelle specifications, generic and implementational Specifications.
- On the other hand if the generic Estelle specifications is available from the Standard Committees, it makes the process of production of a implementational level Estelle specification much easier.
- The interactions exchanged between the modules through the channels can be specified in two styles.

  The first style is called the *Functional Style* [DiDu89], in this style all the different interactions will be enumerated along with all their associated parameters.

  Example:

```
CHANNEL U (UserOfQ931, Q931AsProvider);
 BY UserOfQ931 :
        NL_SetupReq(Param1 : Type1 , Param2 : Type2);
NL_AlertingReq(Param4 : Type4);
        NL_ClearReq(Param1 : Type1, Param3 : Type3);
 BY Q931AsProvider :
        NL_SetupCnf(Param1:Type1, CompletionCode : Typec);
        ...
```

  When an interaction has to be sent, it can be done as shown below:

```
OUTPUT IPX. NL_SetupReq(Channel_1, BASIC);
```

  The above Functional style is simple to use. The OUTPUT statement is similar to a function call with all the parameters being explicitly assigned and passed as arguments(call by value).

The second style is called the *Systems-Programming Style* [DiDu89] and in this style all the different interactions will be interpreted from a common representative pascal record or records.

Example:

```
TYPE UserPrimitives = (NL_SetupReq, NL_AlertingReq, NL_ClearReq);
PrimitiveReqType = RECORD CASE Prim : UserPrimitives OF
                        NL_SetupReq(Param1 : Type1 , Param2 : Type2);
                        NL_AlertingReq(Param4 : Type4);
                        NL_ClearReq(Param1 : Type1, Param3 : Type3);
                    END;

 CHANNEL U (UserOfQ931, Q931AsProvider);
            BY UserOfQ931 :
                        NL_DataReq(UserReqPrim : PrimitiveReqType);
            BY Q931AsProvider :
                        NL_DataCnf(UserCnfPrim : PrimitiveCnfType );
```

When an interaction has to be sent, it should be first initialized as shown below:

```
PrimReqBlock.prim := NL_SetupReq;
PrimReqBlock.param1 := Channel_1;
PrimReqBlock.param2 := BASIC;
OUTPUT IPX.NL_DataReq(PrimReqBlock);
```

The above Systems Programming style is closer to a real implementation. In this style only those parameters which should be sent will be initialized, the rest may contain default values.

With the second style the generic specifications style will be lost, but it is very useful for generic processing of interactions. With the functional style the specifications is easier to read but there will be redundant repetitions of Estelle lines.

In this thesis both the styles are mixed in the specification.

Feature of Estelle for Levels of Refinement

It is possible to refine the generic specification being developed in a stepwise manner making use of Estelle's implicit abstraction levels:

1 Level 1 : Define the structure of the global system, all module definitions and the structure of the interfaces. Keep the bodies of the modules empty.

2 Level 2: Define the first level plus the bodies for all the modules. Keep the functions and procedures empty.

3 Level 3 : Define the second level plus the functions and procedures. The generic specifications is now complete.

Levels of Refinement for Implementational Specifications

4 Level 4 : The channels of the layer under development with its adjacent layers may increase as a result of the introduction of the inter-layer service interactions as explained in chapter IV and chapter VI. There is a need to introduce static and dynamic modules and link them appropriately in order to properly represent the design.

Design of Estelle Specifications for a single layer protocol (Simplified ISDN Q.931)

Even though the attribute of Q.931 module in the ISDN specification is *process* it will consist of nested modules of attribute *activity* only. There is no need for any parallel execution within the ISDN Q.931 specification as it runs on a uni-processor, and to avoid the complexities arising out of asynchronous and synchronous parallelism.

The users of Q.931 Services access the services through the Q.931 Service Access Point (QSAP). The Q.931 Service is provided by a single Q.931 protocol entity, named Q931_Entity, which in turn uses the Data Link Services for the exchange of Messages with its peer and network. Q.931 protocol control layer is treated as one Estelle module which interacts with an upper user of Q.931 module and an underlying Data Link Control provider module. The Data Link Provider is assumed to specify and take care of the underlying ISDN physical layer services. Therefore ISDN specification now consists of three sub-systems namely, ISDN Higher Layer, Q.931 Protocol Control layer and Data Link Control layer sub-systems.

The basic function of ISDN is to allocate data channels and control them for the rest of the life of the call for that particular call. Therefore, it is possible to have as many calls as the number of data channels in the ISDN interface. These calls are obviously accessible to the user above ISDN Q.931 layer. The calls can be identified through the User Connection End Point (UserCEP). In order for Q.931 to control each one of the above calls, separate control blocks should be provided for each individual call. The creation and

manipulation of control blocks for each call will be performed by Q.931 layer, and it is possible to have a separate physical task or a virtual task which controls individual calls. Eg., for ISDN Basic Interface, there are two B channels, therefore two UserCEPs are required to identify the two calls, as a result it is possible to model two Estelle modules to manipulate the control blocks one for each ISDN call.

In the design of Q.931 Specification the above refinement steps will be followed.

The level 1 specification for Q.931 is provided in Appendix A.1 and its static structure is shown in figure 7.2.

The level 2 specification is not provided separately.

It is not in the scope of this thesis to completely specify the level 3 specification with all its functions and procedures. However, important functions and procedures which are essential for the understanding of the simplified Q.931 specification are partially specified or explained with appropriate comments in the level 3 specification in Appendix A.2.

In the design of Level 3 Specification, great amount of difficulty and complexity was realized in the generation of the state transitions (because of the complexity of the ISDN Q.931 protocol) which are internally evaluated once the peer PDU is received. In Estelle the use of *from* and *when* clauses is straight forward. But there are no facilities except the *provided* clause, in order to evaluate the internal criterion for changing the state. Moreover, the provided clause only accepts *boolean* functions, therefore multiple provided clauses were required to model the Q.931 transitions. Many a times the use of *set related operations* was badly felt. The lack of set related operations will lead to writing of complex procedures and functions thus making the specifications hard to read.

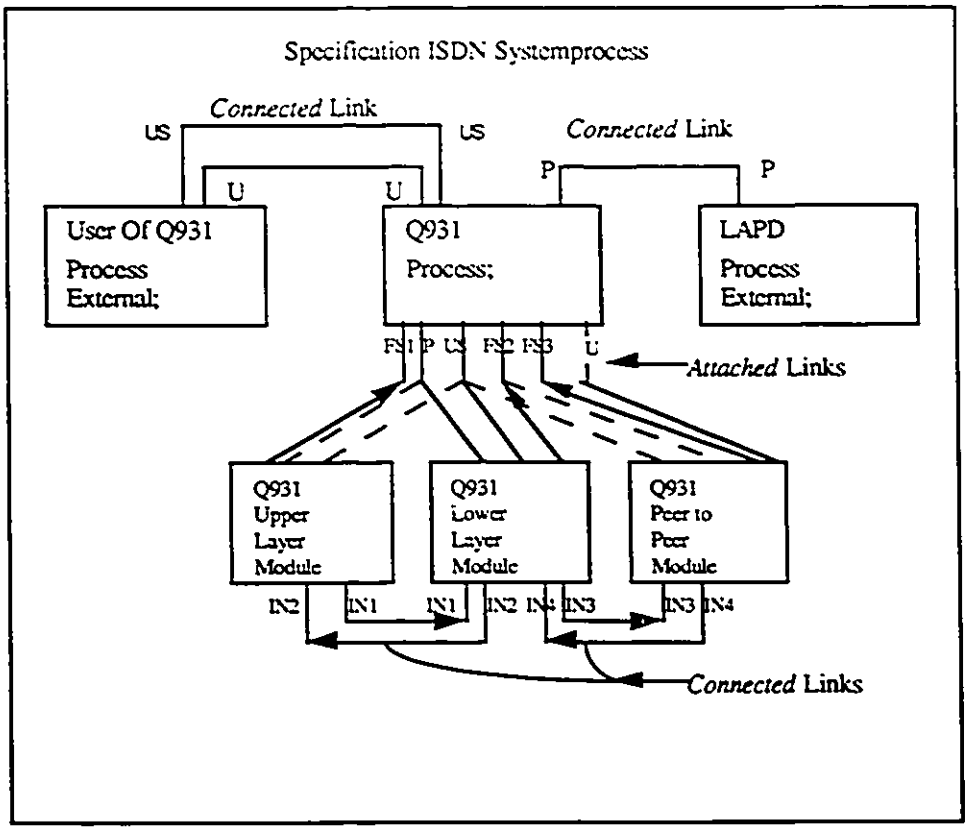
In order to introduce the Q.931 service specifications in the level 3 protocol specifications of Appendix A.2, there was a need to create three additional modules in the body of Q.931 specification as shown in figure 7.3. The existing Q.931 entity module of appendix A.2 handles only the peer-to-peer interactions, therefore it becomes the child of the main Q.931 parent module which will have two other modules as its children (concept of control program of example 3.8.2.E1 to be modelled - solid lines in figure 3.6).The two other modules represent the upper layer interface state machine and the lower layer interface state machine. The parent module will always receive the incoming interactions and will decide which child it belongs to. It will subsequently *create* that child and *attach* U and P interaction points to that child. The child will receive the same interaction as its

parent (because of the attachment) and can send out appropriate interactions through the U or P interaction point.

The following modelling technique for a single layer multi-state-machine communicating system in Estelle is an elegant way for specifying such an implementation-directed design issue, similar strategies may be employed for other design-related issues. A method for specification in Estelle of another typical local design issue is given in [DiDu89].

Sometimes the child does not have an interaction to send out through the IP's attached to its parent. Instead there is a need to pass the locus of control from one internal module to another (dotted lines in figure 3.6). In such a event the child module will *connect* with its sibling and at the same time passes control to the parent module through an *export* variable The parent will decide based on the above export variable or IP whether an internal event is generated or not. Information related to the internal event such as type of internal event, source state machine and destination state machine can all be sent a exported variables or alternatively specified in a global data structure. The alternative method is closer to implementation. Based on the information of the internal event the control module will *attach* the destination module to its (parent's) IP's, namely U and P. This attachment is done because as a result of the *connection* of the sibling modules the destination sibling module may send interactions to adjacent layer which can interact only through the U and P IP's.

This process is repeated until the parent module gets an exported variable or internal interaction indicating there are no more internal events. The partial implementational specification is shown in Appendix A.3. Appendix A.3 is meant only to give a general idea to introduction of just one of many possible designs into the formal specifications. With the inclusion of internal events the U and S state machines of diagram 4.2 will change, for example the change in U state machine is given in figure 7.4 which is used in appendix A.3. The appendix A.2 and A.3 are just the fist versions which can be further developed until they are compilable within a reputable Estelle compiler to produce ISDN Q.931 protocol code.



NOTE: The dotted lines indicate the dynamic connection not established at this time.

The solid lines indicate the present established dynamic or static connections.

Only P and US IPs are Dynamic.

FS1..FS3 are IPs with interactions from son to parent module only. They enable the parent module to decide about the dynamic links.

IPs from IN1 to IN4 are meant for exchange of internal primitives between modules.

Figure 7.3 A Dynamic Instance of Q931 Process and Lower Layer State Machine Module

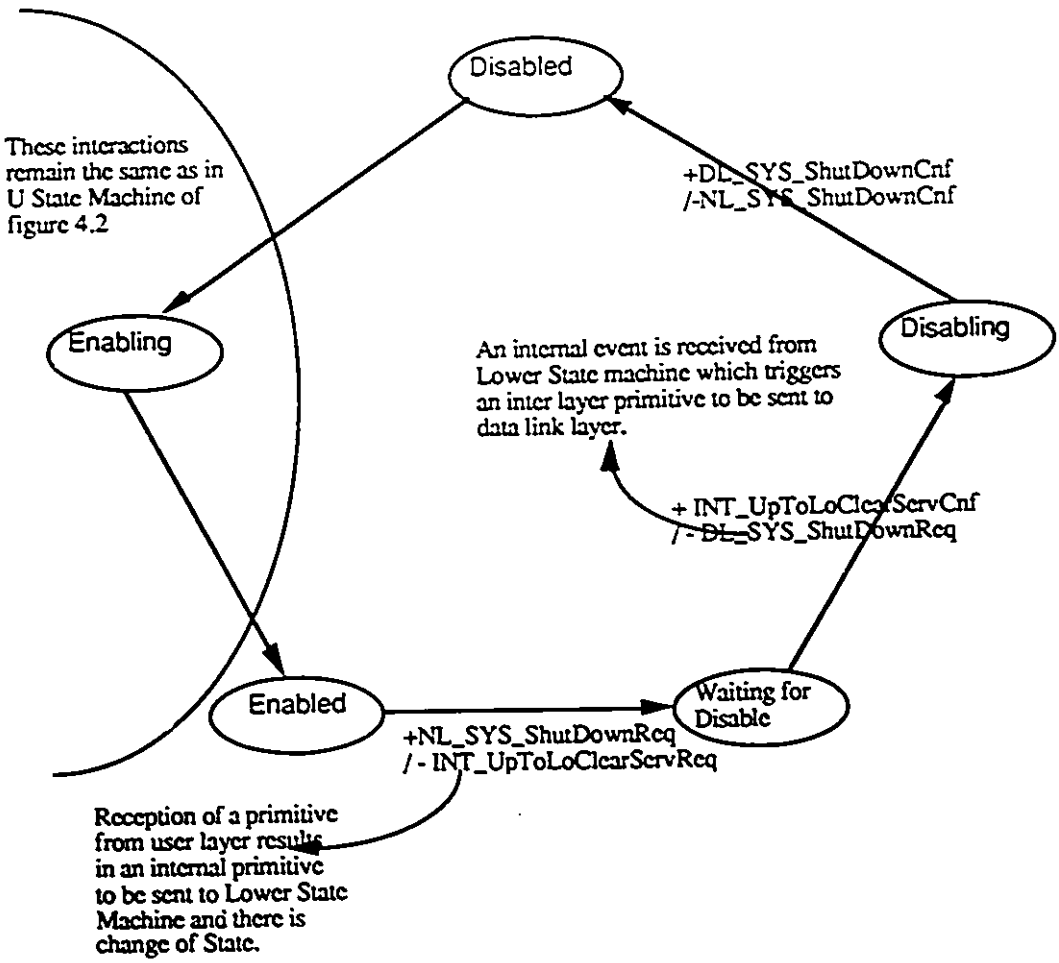


Figure 7.4 U State Machine With Internal Primitives

7.3 To Specify the External Context Environment

External context environment, E, for ISDN is a function of Constants Identifiers, Type Identifiers, Channel identifiers and Procedure-and-Function Identifiers, E = (Ce, Te, Che, PFe). The intention of the following paragraphs is to provide a flare for the parameters of Q.931 which need to be specified in Estelle in order to make it comprehensible and complete. These parameters will outline some of the essential factors in the selection of the above environment.

Constant-Identifiers (Ce) required:

The ISDN Systemprocess will co-ordinate the communication between its processes.

As we know from ISDN overview, basic and primary ISDN supports multiple B channels. According to ISDN CCITT I.451 every ISDN call will be identified by a

unique call reference and each call will be responsible for a single B channel. Therefore, we should have a constant which depicts the maximum number of users of Q.931 protocol control layer which is equivalent to 2 for Basic interface.

Obviously, the Higher Layer module should request Q.931 for a service from Q.931. For example in order to setup an ISDN call the Higher Layer should issue NL_SetupReq as indicated in figure 6.5. As a result Q.931 will build a SETUP message and transmit it to the peer. In order for SETUP message to be constructed, a number of values are required as shown in the SETUP message and its Information Elements in TABLE 25/Q.931 of recommendation CCITT I.451/Q.931 [CCIT3]. The details of such values are not specified in this thesis.

Type-Identifiers(Te) Required:

The global values required by Q.931 process which should be specified by its user are not always of type which can be represented as Constants. Some of the required fields may be of composite types. These types have to be first identified in the type-definition part of E.

For eg., the values for peer address which needs to be transmitted in the Called Party Address IE in the SETUP message will be specified by the *user of Q.931* process as a parameter in the *Connect.Req (NL_SetupReq)*. The type of this address is a complex one.

In order for Q.931 process to know what are the possible primitives which can be issued by the User of Q.931, a composite type representing all the primitive identifiers should be provided. The type for each identifier should be provided.

Channel-Identifiers(Che) Required:

Since OSI architecture is followed for the specification of Q.931 process , this layer will have two interaction points, one with the user layer and the other with the provider layer. Through these interaction points the processes will exchange the inter-layer primitives. Thus the total interactions at the two interactions points of Q.931 is a Cartesian product of Identifiers with the Primitives.

The interactions received by Q.931 is appended to an *unbounded* (once arrived interactions are always accepted, whatever it be) FIFO queue associated with an interaction point.

In our specification of Q.931, there is no synchronous mechanism for the exchange of interactions, which means any task can send an interaction which will be added to the specific channel tail. The Q.931 process will extract each primitive from the

queue and process it. It will not go for the next element in the queue until the transitions as a result of the previous primitive are completed (*atomic behavior of transitions*).

The channel definitions and some generic parameters are shown in appendix A.1

7.4 Evaluation And Summary of Implementation-directed Specification of Q.931

The author's experience in writing the specification for a subset of ISDN Q.931 brings the following complexities to the attention. As the available ISDN Q.931 specification was not in Estelle, it was essential to thoroughly understand the Q.931 specification available in English. The aim of the thesis is to assist the process of automatic generation of protocol code, therefore it was essential to induce the implementation-directed design of Q.931 into the specification. In order to generate implementation-directed specification of any protocol it is necessary to understand that protocol completely and should have knowledge of design of implementation of protocols based on generic and local issues. It is also necessary to have a thorough knowledge of Estelle especially the ability to model certain complex situations using a better specification approach. It is possible to specify a protocol in many ways, likewise it is possible to have multiple implementation-directed specifications based on the choice of design.

The main difficulties in specifying generic ISDN Q.931 protocol are covered in section 7.4.1 and 7.4.2. The generic ISDN Q.931 protocol is later changed into a prototype for implementation-directed specification by adding the services specifications as well sample standard system interface calls in appendix A.3. It is noticed from appendix A.3 that, once the service specifications are specified based on our design approach (chapter 3 and chapter 6), it is possible to easily specify such an inter-layer state machine in Estelle. It is also possible to specify the control program for such inter-layer state machines though with some difficulty.

The above appendices are only representative examples for generic and design-directed specifications of ISDN Q.931. The above specifications makes use of many partially defined system interface and function calls. However, it is possible to run the above specifications through standard Estelle compilers if the target operating system offers a standard interface and all the functions of above specifications are fully specified. Such a run will also detect many Estelle syntactic errors. The aim of the thesis is to demonstrate the design process of service specifications and system interactions and the ability to specify them using Estelle resulting in an implementation-directed specification. In this process number of difficulties at various phases of protocol engineering process are identified. The author has contributed towards the protocol engineering process framework and

methodology and it is not necessary to thoroughly verify and test the resultant Estelle specifications.

It is possible to run the generated partial ISDN specification with a few modifications to suit the NIST Estelle compiler which apparently does not support complete Pascal language at this stage. Moreover, the Estelle compiler is not used because the generated Estelle specification represents only a subset of ISDN with major functions being represented only as comments, also there is no standard interface available and the ISDN messages themselves are not specified explicitly in either ASN.1 or Estelle. However, this specification lays a basis for future modifications and enhancements.

The Estelle module concept permits easy modelling of communicating processes. It is not necessary all the Estelle modules will translate into multiple communicating processes in the executable C code. For further understanding on the mapping of various Estelle features refer to [Amalu87], [VoLa88].

7.4.1 Problems Encountered in writing Estelle Specification

The following potential demerits of Estelle were obtained from a survey of literature [Cour87], [DiDu89], [Phalip89] and the experience of the author in writing the ISDN Q.931 specifications.

1. Estelle makes use of Pascal which is an implementation language. Hence the Estelle Specifications are implementation oriented. A detailed discussion of features of Pascal which may lead to over specification of Estelle specifications is in [Phalip89].
2. Writing of Estelle specifications of an OSI system [Zimm80] by making use of the attributes of Estelle modules (systemactivity and systemprocess), depends on the software and/or hardware division of the OSI system. Since a generic OSI specifications is not meant for a specific implementational choice. This leads to over specification in Estelle.
3. According to [Cour87], *Priorities* of Estelle transitions are useful in an implementation but will lead to problems in a formal specifications (such as relationship between the priority clause and the delay clause of Estelle). One of the identified problem is that, a "priority" clause is redundant as its scope is only local to a module, and that there are always alternate means of writing a specifications, and the "priority" clause is overridden by the "parent/child" priority. But the author suggests the use of priority class for prioritizing the errors detected in incoming PDU's, similar suggestions are made in [DiDu89].

4. For direct or indirect interactions between modules of a specifications Estelle imposes a particular implementation scheme (FIFO queue), while it is possible to implement other valid schemes (for eg., a simple procedure call). An Estelle interaction point is representative of a Service Access Point between adjacent layers of OSI architecture. The entities or functional groups within the same layer of an OSI architecture are specified as nested modules within a "subsystem" for that layer. Thus the modules within a subsystem will also have interaction points which is an improper concept of Service Access Point.
5. Estelle specifications normally have combined *service specifications* and the *protocol specifications* for any layer. The addition of service specifications leads to problems [Cour87] such as

   Collision of service primitives,

   Backpressure flow control,

   Conflicts in the allocation of connection endpoints.

   The author has suggested the development and use of a *standard system interface* for the protocols of OSI layers in chapter IV. This standard system interface can restrict the protocol to send/receive a fixed number of primitives, alternatively the protocol itself can calculate this limit and notify the system interface. The system interface will now control Backpressure if the user layer sends more requests than identified before. Similarly by following rules of queue priorities between the adjacent layers which are imposed by the system interface and by having a unique identifier for each primitive it is possible to eliminate the problems arising out of Service Primitive Collision.
6. Estelle doesn't provide a means to solve the problem of Conflicts in the allocation of connection endpoints. The author has suggested a simple mechanism of solving the above problem as in section 3.8.1.1 by having two connection end point identifiers. This problem along with many problems discussed above are resolved in [Cour87] by a proposed synchronized communication mechanism [Rendez-vous] incorporated in Estelle* and the elimination of Estelle clauses such as "priority" and "delay" and concept of "system" and related asynchronous parallelism [through message queues]. The inclusion of Rendez-Vous mechanism within the Estelle standard (IS9074) is currently being investigated within the ISO FDT group under the framework of question 48.5 (JTC1/SC21/WG1).
7. Estelle channels are bi-directional but asymmetric. Each end point is associated with a specific role and the two end-points are associated opposite roles. As a result of this asymmetry it is not possible to have two or more instances of the same module

communicating with over the same channel. Estelle's solution to this problem leads to a redundant definition of the module header and body for the same module or duplication of the Interaction Points of the channel (hence channels). An example is provided in [DiDu89].

8. Availability of Estelle Tools [AnAm87] such as syntax driven editor and debugger [ChLe89], window based graphics interface [NeAm89] will assist in development of formal specification of a protocol.

7.4.2 Features Lacking in Estelle

After the experience gained by the author in the writing of the subset of ISDN Q.931 Specification the author thinks that availability of certain features in Estelle will be useful for writing of precise Estelle Specifications.The author also finds the observations pointed out in [DiDu89] valid regarding the features lacking in Estelle.

1. From [DiDu89], it is possible to create dynamically the modules from within the execution of transitions of a parent module. But the Interaction Points themselves for the modules should be defined statically. Therefore, there is a fully dynamic allocation and release of resources of a module but not for the links between them.
2. From [DiDu89], a generic WHEN is useful for situations when every incoming message should be treated the same way. The present Estelle specifications will become very long if the same process is repeated for every enumerated incoming message.This problem is noticeable in Q.931 specification. This problem is also experienced outside of a generic WHEN as in point 3.
3. The author has found that many a times the incoming set of peer messages or inter-layer primitives(TRANS-SET) is very large. There will be many subsets of TRANS_SET, which will result in certain common action to be taken within the transition part of a module body. Even though Estelle provides the set variable stateset_which is indicative of a subset of TRANS_SET, the syntax of Estelle Transition Clauses don't permit the usage of Pascal Set operators within the transition body according to [ISO3]. The author has searched the available literature on Estelle and noticed that there is no usage of set operators in the "when" clause of Estelle Transitions. More over there are no suggestions either to overcome the problem of redundant repetition of transitions in the transition part of a module for members of TRANS-SET subsets which share the common properties. However, Estelle permits the usage of Pascal Set operators in the action part of a transition which consists of pure pascal assignments, functions or procedure calls. Also, Estelle provides a mechanism for searching by the parent module of its children

modules by making use of special quantifiers (EXIST, SUCHTHAT, FORONE etc.,) which are not found in Pascal[Phalip89]. The author is suggesting the inclusion of Pascal Set operators in the "when" clause of Estelle Transitions or certain special quantifiers.

4. The author has also noticed that, Estelle has the concept of having the "Specific Type" of modules. Instances of certain types of modules are created by using the *modvar* and *init* key words of Estelle. There is no such concept of "Types Of Channels". If there is a need for two channels to be of same type but are in no way linked (neither connected nor attached directly or indirectly) then the Channel Definitions and their roles should be repeated. The author is suggesting the availability of concept of declaring "Types Of Channels" such that instances of channels can be created whenever required.
5. Estelle permits the exchange of only one interaction through two interaction points of opposite roles of a channel between any two modules. There is no facility to broadcast the same interactions to multiple modules through multiple channels which is required in specifying many Local Area Networks and in LAPD which is the data link control protocol of ISDN. The lack of this feature is discussed in [BaCo90] and [SaCo89].

Chapter 8.

8.1 Conclusion & Suggestions for Future Work

Automation of protocol code generation [SiCh89, VoLa88, BoGe87] was shown to be possible based on the use of formal specification techniques such as Estelle, LOTOS and SDL.

The above FDTs are most practical for specifying the semantic part of a protocol and their use therefore tends to result in only partial automation (as explained in section 3.5) of a protocol specification (the exchange of PDUs, timers, and the action part of the transitions).However, a protocol specification is complete only if both the semantic and syntactic parts are specified. The syntactic part consists of the format or structure of the PDUs. A standard abstract syntax notation is available which is used to specify the PDUs. ASN.1 is widely used, for representing the data structures of application layer PDUs only. But this author has presented arguments in section 3.5 supporting the use of ASN.1 even for lower layers, which are the relevant ones for this thesis.

Tools are available for formal protocol specification languages and recently for the standard abstract notation of PDUs (refer to [Boch90b] for a very useful survey of available tools). There seems to be a research trend to merge the implementation code generated by tools for protocol specification languages with the code generated by tools for abstract syntax notation for the PDUs.

However further research is needed to decide the ideal stage to merge the specifications of a protocol with those of the abstract syntax notation — it is certainly reasonable to merge the above specifications before producing the executable code. Hence, tools are needed to process an FDT specification language and the abstract syntax notation at the same time. A method of such merging is proposed in [Boch90b]. It is a very complex problem to share the structures between Estelle and ASN.1 specifications so that the resulting implementation code also has shared structures. There is a research need to

develop tools which will evaluate the correctness of such a composite specification before it is used for implementation.

The merging of implementation code which results from the Estelle Specification and ASN.1 coder and decoder is also complex. It is difficult to share the implementation structures as the implementation code is generated by two independent processes. Sometimes the structures of PDUs are part of a larger control block within the Protocol Transitions. Moreover, the protocol transitions, action routines and the coder and decoder parameters are closely bounded and the code itself is interleaved.

According to this author, a practical communications protocol entity consists of an implementation of protocol semantics, a parser which knows the syntax of the PDUs and hence can code and decode the PDUs, the service interface, and the system interface. The first two are explained in the above paragraphs and discussed in the research literature [Boch90b]. The service interface specifications have been studied but there are only few concrete examples which have been published. However, much of this thesis was devoted to formulating and applying principles to generate service specifications for a real world protocol. The only remaining component in the generation of the entire protocol code is the system interface.

According to a survey the author has performed on the existing literature, the system interface issues are generally ignored. Essentially, *there is only so much a protocol requires from an operating system* which is justified by the analysis of operating system services, presented in this thesis. Thus, use of a *standard system interface* given in [AlFe90] [Koen90] and suggested in [Martin88] should become part of formal implementionl specification to generate the system dependant code as an integral part of the automatically generated protocol code.

A global diagram, envisioned for generating complete protocol code automatically is shown in figure 8.1. Based on the types of tools developed for ASN.1 and Estelle Specifications the components shown in figure 8.1 will vary. A more detailed figure for automated protocol code generation (except the system interface) is given in [Boch90b]. Developing of a prototype of a real-world protocol based on figure 8.1 should be carried out to test the cost-effect issues of this approach.

One of the aims of this thesis has been to capture complete service specifications into a formal specification language. It is suggested in this thesis that the pseudo code development stage be replaced by an implementation directed specification development stage using FDTs. A specific design approach to arrive at the service specifications, which essentially consists of a user layer and provider layer interface state machines, is presented. The user, provider and the peer-to-peer state machines are in turn controlled by a control

program, which decides upon the transfer of locus of control to a particular state machine. Introducing any design involving multiple state machines will require local validation before a peer-to-peer validation is started. A potential research focus for such validation is in the next section.

This thesis has presented basis for the design of service specifications based on many generic factors and local support environment. This service specification design approach will assist in the creation of a *Service Creation Environment* which will consist of *Service Logic Editor* (SLED) and *Service Analysis Logic Tester* (SALT) such as the one developed by Bellcore [Martin88].

In order to specify any protocol using a formal specification language, it is necessary to have a command of the specification language. Complex situations are encountered during specification of a real protocol. It is beneficial to have a *guide book* providing specification examples for generic functions of protocols and a collection of such complex situations modelled previously during experience with a protocol. Much research needs to be done to develop such a guide book for different FDTs.

There is also a traceability issue between the generated code and the corresponding Formal Specifications. Formal Specification based implementation can vary, but the generated executable code should perform the same functions. Moreover, use of non-standard Estelle specification mechanisms such as Rendez-Vous mechanism should also result in an executable code which is functionally identical. There is a need for an analysis of the traceability of formal Estelle constructs such as Modules, Synchronization mechanisms, etc. to corresponding output C code.

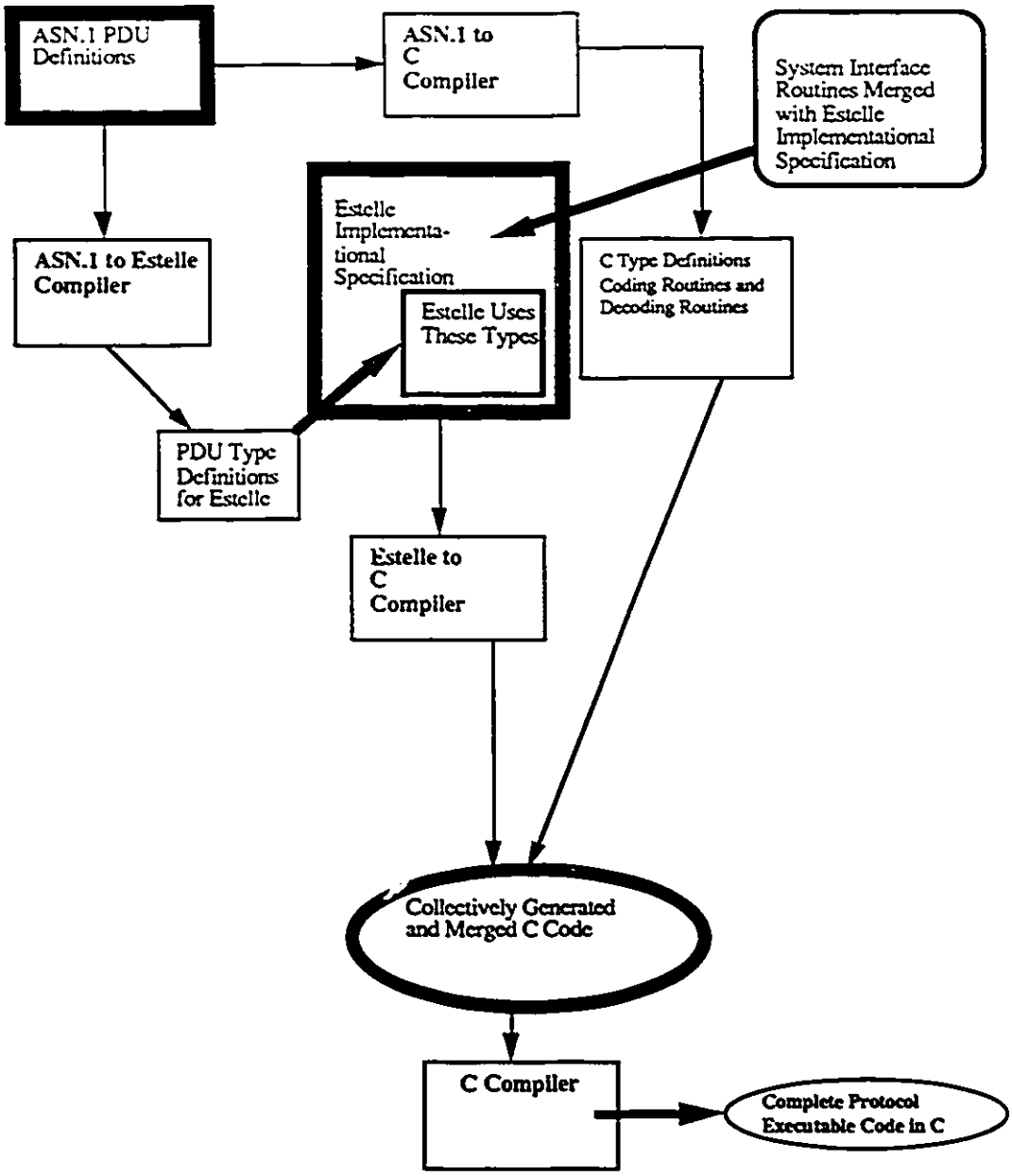


Figure 8.1 A Possible Scenario For Complete Protocol Code Generation Automatically

8.2 Validation Of Service Specifications

This thesis suggests implementing a *protocol specification* and its *Service Specifications* as multiple state machines, in section 3.7.1. It is also indicated that there will be a control program guiding the incoming events to the respective state machine, in section 3.7.3 and example 3.8.2.E1.

The author believes that even though the validation techniques themselves [Hail82], [West86], [RuWe82], [Pehr89] can be applied to validate multiple communicating state machines, the validation in general is performed between two peer-to-peer communicating state machines. While performing validation of peer-to-peer state machines it is reasonable to ignore the inter-layer interactions. But the inter-layer interactions themselves are between adjacent communicating state machines and violating protocol properties here can terminate peer-to-peer communication. Therefore, the author believes it is important to validate the peer-to-peer state machine in an implementation together with its inter-layer interactions. Validating an implementational layer will therefore consist of these additional issues, along with all those which existed previously:

1. There is communication between the peer-to-peer state machines as before.
2. Before the peer-to-peer state machine is invoked, there is interaction between the state machines of the adjacent layers.
3. The state machines are not completely independent. Reception of a primitive does not result in its reaching of a particular state machine directly, instead it is determined by the control program.
4. The control program will transfer the locus of control to one of the state machines based upon the previous state of a particular state machine or a combination of the states of multiple state machines (global state of a protocol layer) and the incoming primitive.
5. Thus locus of control is a function of two mechanisms, namely, the incoming event and the history of states of a state machine.

The state machine model for the services provided to a layer defines the behavior of all the state machines of that layer. Since the primitives arrive from various sources (user layer, server layer, error recording utilities, performance recording utilities etc.) the state machines should be able to handle any non-deterministic arrival of an event. The potential for arrival of every event in every state of every state machine should be considered, and suitable action routines should be written in order to perform a valid transition. This may result in a state explosion problem. However, further research can be done to trim the

reachability tree by making use of factors such as logical justification behind the locus of control provided by the control program.

In Summary it is the authors hope that a protocol specification which is semantically and syntactically complete (using FDTs) will be available to protocol engineers who can modify such an specification into an implementation-directed specification which can result in the generation of completely operational protocol code.

References

[AlFe90] Aleem Syed A., Fekete Steve I., "A standard system interface for protocols", to be published.

[Amalu87] Amalu Charles O., "A Tool for Semi-Automated Prototyping of Communication Protocols", Master's Thesis, University Of Ottawa, Ottawa 1987.

[Ansa82] Ansart J.P., Rafiq O., Chari V., "PDIL - Protocol Description and Implementation Language", Proc. IFIP WG.G.1 2nd Int. Workshop on Protocol Specification, Testing and Verification -California, May 1982.

[Ansa83] Ansart J.P., Chari V., Simon D., "From Formal Description to Automated Implementation Using PDIL", Protocol specification, Testing and Verification, III, IFIP 1983, pp.381-390.

[AnAm87] Ansart J.P., Amer P., et al., "Software Tools for Estelle", IFIP International Workshop on Protocol Specification, Testing and Verification, VI, Quebec, North-Holland, Amsterdam, 1987.

[ASN.1] ISO IS 8824, "Information Processing - Open Systems Interconnection - specification of Abstract Syntax Notation One (ASN.1).

[AyCo82] Ayache J.M., Courtiat J.P., "LC/1, A specification and Implementation Language for Protocols", Protocol specification, Testing and Verification, III, IFIP 1983, pp.333-346.

[BaCo90] Barretto M. L., Courtiat J. P., Saqui-Sannes, "Experience in Using Estelle* for the Specification and Verification of a Fieldbus Protocol: FIP,"Proceedings of the IFIP TC6 International Conference on Computer Networking COMNET'90, Budapest, Hungary, May 1990.

[Bell83] Bell Laboratories, Volume 1, "UNIX Programmers Manual Revised and Expanded Version, Bell Laboratories, 1983.

[Boch78] Bochmann G.V., "Finite State Description of Communication Protocols", Computer Networks, vol. 2, no. 4/5, Sept-Oct 1978, pp361-372.

[Boch80] Bochman G.V., "Semiautomatic implementation of communication protocols", IEEE Transactions on communications, Com-28, No.4, April 1980.

[Boch87] Bochman G.V., "Usage of Protocol Development Tools: the results of a survey", IFIP,1987.

[Boch90] Bochmann, G.V., "Protocol Specification for OSI", Computer Networks and ISDN Systems", Jan 1990.

[Boch90b] Bochmann G.V., et al., "Implementation support tools for OSI application layer protocols", University of Montreal, Technical Report #720, March 1990.

[BoBr87] Bolognesi T., Brinksma E., "ISO Specification Language LOTOS," Computer Networks and ISDN Systems, Vol. 14, 1987.

[Boch88] Bochmann G.V., "Specifications of a simplified Transport Protocol", Technical Report, Publication No.623a, University of Montreal, June 1988.

[BoDe86] Bochmann G. V., Deslaurier M., and Bessette S., "Application layer protocol testing and ASN.1 support tools", Proc. of IEEE GLOBECOM Conf., Houston, Dec 1986.

[BoGe87] Bochmann G.V., et al., "Semiautomatic Implementation of Communication Protocols", IEEE Trans. on Software Engineering, Vol. SE-13, No.9, Sept 87.

[BoGo86] Bochman G.v., Gotzhan R., "Deriving Protocol Specifications from Service Specifications," Proc. ACM SIGCOM Symposium, pp. 148–156, 1986.

[BoJo79] Bochman G.v., Joachim Tankoano., "Development and Structure of an X.25 Implementation", IEEE Transactions on Software Engineering, Vol. SE-5. No.5 1979.

[BoLo90] Bochman G.v., Logrippo Luigi., Behcet Sarikaya., "Formal Specifications for Protocols: Issues and Experiences", Proceedings of the IFIP TC6 International Conference on Computer Networking COMNET'90, Budapest, Hungary, May 1990.

[BoRa83] Bochman G.v., Raynal Michel., "Structured Specification of Communicating Systems", IEEE Transactions on Computers, vol. C-32, No. 2, Feb 1983.

[BoSu80] Bochman G.v., Sunshine C. A., "Formal Methods in Communication Protocol Design," IEEE Trans. on Communications, Vol. Com-28, No. 4, pp. 624–631, April 1980.

[BrZa80] Brand D and Zafiropulo., "Synthesis of protocols for unlimited number of processes. In proc. Trends and Applications: 1980 Computer Network Protocols, NBS, Gaithersberg, MD, 1980.

[BrZa83] Brand D., Zafiropulo P., "On communicating Finite State Machines," J. ACM, Vol. 30, No. 2, April 1983.

[BuDe87] Budkowski S., Dembinski P., "An Introduction to Estelle: a Specification Language for Distributed Systems", Proceedings IFIP , Zurich, 1987.

[CaDu86] Castanet R., Dupeux A., Guitton P., "ADA a well suited language for specification and implementation of protocols," Protocol specification, Testing and Verification, V, IFIP 1986.

[CaCo88] Cameron E. Jane, et al., "The IC* Model of Parallel Computation and Programming Environment", IEEE Transactions on Software Engineering, Vol. 14, No. 3, 1988.

[CCIT1] CCITT "Recommendation Q.920, ISDN User-Network Interface Data Link Layer - General Aspects"

[CCIT2] CCITT "Recommendation I.440/I.441 Q.921, ISDN User-Network Interface Data Link Layer Specification", red book, Geneva, 1985.

[CCIT3] CCITT, Recommendation I.450/451 or Q.931, ISDN User-Network Interface layer 3, red book, Geneva, 1985.

[CCIT4] CCITT Red Book Fascicle VI.11 - "Functional Specification and Description Language (SDL)", Recommendations Z.101-Z.104, Geneva, 1985.

[CCIT5] CCITT Recommendation X.25, Interface Between Data Terminal Equipment (DTE) and Data Circuit-terminating Equipment (DCE) for Terminals operating in the Packet Mode on Public Data Networks (Revised), 1984.

[Chong86]Chong H.Y., "Software Development and Implementation of NBS Class-4 Transport Protocol" , Computer Networks and ISDN Systems 11, 1986.

[Chow78] Chow T. S., "Testing Software Design Modelled by Finite State Machines," IEEE Trans. on Soft. Eng., Vol. SE-4, No. 3, pp. 178–187, May 1978.

[ChGo84] Chow C.H., Gouda M.G.,"An Excercise in constructing multi-phase communication protocols", in Proc. ACM SIGCOMM'84 Symp., June 1984.

[ChGo85] Chow C. H., Gouda M. G., Lam S., "A Discipline for Constructing Multi-Phase Communication Protocols," ACM Trans. on Com. Sys., Vol. 3, No. 4, pp 315–343, Nov 1985.

[ChLe89] Chari V., et al., "An Estelle Simulator/Debugger Tool (edb), " The Formal Description Technique Estelle, Diaz, Ansart, Courtiat, Azema, Chari (eds.), North-Holland, Amsterdam, 1989, 381-396.

[ChMi86] Choi T. Y., Miller R.E., "Protocol Analysis and Synthesis by Structured Partitions," Computer Networks and ISDN Systems, Vol. 11, No. 5, pp. 367–381, May 1986.

[ChSa89] Cheung To-Yat., Sablatash Mike., "A Functional Network Model for Analytical File Management in ISDN Systems From Generalization of Videotex Systems," Computer Networks and ISDN Systems 16, pp.299-310, 1989.

[Cour87] Courtiat J.P., "How could Estelle become a better FDT?", Protocol Specification, Testing and Verification, VII, IFIP 1987.

[Cour88] Courtiat J.P., "Estelle*: A powerful dialect of Estelle for OSI Protocol Description" , Protocol Specification, Testing and Verification, VIII, IFIP 1988.

[Datapro89] Data Communications, C07-500-301 Standards, Datapro Research, 1989 McGraw-Hill Incorporated.

[DhKo86] Dhas C.R., Konangi V.K., "X.25: an Interface to Public Packet Networks", IEEE Communications Magazine, Vol.24, No.9, Sept 1986.

[Diaz82] Diaz M., "Modelling and Analysis of Communication and Cooperation Protocols using Petri Net Based Models". Computer Networks, 6(6), 1982.

[DiDu89] Diaz M., Dufau Jean., Groz Roland., "Experiences Using Estelle Within SEDOS Estelle Demonstrator", FORTE'89, 2nd International Conference on FDTs Vancouver, 1989.

[Diet90] Dieter Carl., "Functional and Performance Prototyping concepts based on SDL", Proceedings of the IFIP TC6 International Conference on Computer Networking COMNET'90, Budapest, Hungary, May 1990.

[DiPi83] Disckson Gary J., Chazal Pierre E., "Status of CCITT Description Techniques and Application to Protocol Specification", Proceedings of the IEEE, Vol. 71, No.12, Dec 1983.

[ElKu90] Zvi Har'El., Kurshan Robert P., "Software for Analytical Development of Communications Protocols." AT&T Technical Journal, Jan/Feb 1990.

[ExPo82] Exel Matija., et al., "SL1 Language -A Specification and Design Tool for Switching Systems Software Development", IEEE Transactions on Communications, Vol. Com-30, No.6., June 1982, pp. 1356-1362.

[FeSa90] Forghani B., Sarikaya B., et al., "An Estelle Based Test Generation Tool for Modular Specification," Proceedings of the IFIP TC6 International Conference on Computer Networking COMNET'90, Budapest, Hungary, May 1990.

[GaHa88] Gantenbein D., Hauser R., Mumprecht E., "Implementation of the OSI Transport Service in a Heterogeneous Environment," Pages 215-241 in HECTOR, Vol II: Basic Projects, Springer Verlag, 1988.

[GoYu84] Gouda M.G., Yu Y.T., "Synthesis of communicating finite state machines with guaranteed progress. IEEE Transactions on Communications, COM-32(7):pp. 779-788, July 1984.

[Hail82] Hailpern B., "Verifying Concurrent Processes Using Temporal Logic", Lecture Notes in Computer Science 129 (Springer, Berlin, 1982).

[Hail83] Hailpern B., "Specifying and Verifying Protocols Represented as Abstract Programs", Computer Networks and Protocols, P.E.Green (Ed.)., May 1983, pp. 607-624, Plenum Press.

[HaOw83] Hailpern B. T., and Owicki S. S., "Modular Verification of Computer Communications Protocols", IEEE Trans. on Comm., Vol. COM-31, No.1 January 1983, pp. 56-58.

[HeRa78] Hertweck, F., et. al., "X25 Based Process - Process Communication", Computer Networks 2 1978.

[Hoar85] Hoar, C.A.R., Communicating Sequential Processes, Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[IBM74] IBM Corp., Guide to PL/S II, IBM Form No. GC28-6794-0, 1974.

[IBM79] IBM Corp., Programming Language for Distributed Systems Reference (PL/DS), IBM Form No.SC27-0446-0, 1979.

[IBM80] IBM Corp., "Systems Network Architecture Format and Protocol Reference Manual: Architecture Logic", IBM Form No. SC30-3112-2, 1980.

[ISO1] ISO "Information Processing Systems - Open Systems Interconnection - Basic Reference Model," DP7498, (1983).

[ISO2] ISO "Information Processing Systems - Open System Interconnection - Service Conventions", ISO/TC97/SC16/N1646.

[ISO3] ISO "Estelle—A Formal Description Technique Based on an Extended State Transition Model," DP9074, Oct. 1986.

[ISO4] ISO "OSI Conformance Testing Methodology and Framework, Part 1: General Concepts," DP9646–1, Sept. 1986 (Egham).

[ISO5] ISO "The Tree and Tabular Combined Notation," DP9646–3, July 1988 (Sweden).

[ISO6] ISO "OSI Conformance Testing Methodology and Framework, Part 4: Test Realization," DP9646–4, May 1988.

[ISO7] ISO "OSI Conformance Testing Methodology and Framework, Part 5: Requirements on Test Laboratories and Clients for the Conformance Assessment Process," DP9646–5, May 1988.

[ISO8] ISO "LOTOS—A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour," DP8807, March 1985.

[ISO9] ISO "Guidelines for the Application of Estelle, LOTOS and SDL," JCT1/SC21 N2549, March 1988.

[JaBo83] Jard C. and Bochmann G.V., "An Approach to Testing Specifications", J. System Software, Vol. 3, No.4 pp.315-323, Dec. 1983.

[Kanu86] Kanungo B. et al, "A Useful FSM Representation for Test Suite Design and Development," Sixth International Workshop on Protocol Specification, Testing and Verification, pp 163–176, June 1986.

[KeRi84] Kernighan B. W. and Ritchie D.M., "The UNIX Programming Environment, Prentice-Hall Software Series(1984).

[Koen90] Koenig Hartmut., "Experience in Computer-Aided Protocol Implementation", "Proceedings of the IFIP TC6 International Conference on Computer Networking COMNET'90, Budapest, Hungary, May 1990.

[KrKr87] Krishnakumar A.S., Krishnamurthy B., Sabnani K., "Translation of Formal Protocol Specification to VLSI Design," Protocol specification, Testing and Verification, IFIP 1987.

[Lai89] Lai Wai Sum., "Packet Mode Services: from X.25 to frame relaying", 'Packet forwarding', IEEE Communications Magazine, Vol 26 No 7, 1988.

[Linn88] Linn, R.J., Favreau, J.P., "Application of Formal Description Techniques to the Specification of Distributed Test Systems", Proceedings of INFOCOM88, IEEE, 1988.

[Linn86] Linn, Richard J., " The Features and Facilities of Estelle", Protocol Specification, Testing and Verification, V, North Holland, IFIP 1986, pp. 271-296.

[LSL1] Rinaldy E.A., Strebendt R.E., "LSL-Local User's Manual-Version 2.0", Internal AT&T Document.

[Logrip83] Logrippo Luigi., "Constructive and Executable Specifications of Protocol Services by using Abstract Data Types and Finite State Transducers," Protocol Specification, Verification and Testing III, IFIP 1983.

[Martin88] Martin R.L., "Future Telecommunications Services" Proc. GLOBECOM 88, June 1988.

[Miller90] Miller Raymond E., "Protocol Verification: The First Ten Years, The Next Ten Years; Some Personal Observations", Invited Paper, X International IFIP WG6.1 symposium on protocol Specification, Testing and Verification, Ottawa, June 1990.

[Miln80] Milner R., "A Calculus of Communicating Systems," Springer-Verlag, Berlin, 1980.

[NeAm89]New D., Amer P., "Adding Graphics and Animation to Estelle," IX IFIP International Symposium on Protocol Specification, Testing and Verification," Enschede, The Netherlands, June 1989.

[NET81] Networks From The User's Point Of View., L. Csaba, T.Szentivanyi, K.Tarnay (editors), North-Holland Publishing Company, IFIP, 1981.

[Nash83] Nash S. C., "Automated Implementation of SNA communication protocols", Proc. IEEE International Conference on Communication, Page 1316-1322, June 19-22 1983.

[OSIstat] "Status of OSI (and related) Standards", Computer Communications Review, SIGCOMM, ACM Press, October 1988.

[Pascal1] International Organization for Standardization, Programming Language-Pascal. ISO TC 97/SC 5/WG 4, International Standard 7185, 1983.

[Pehr89] Pehrson Bjorn., "Protocol Verification for OSI", Computer Networks and ISDN Systems 1989.

[Phalip89] Phalippou M., "Functional Specification for an ISDN Switching System: an experience using Estelle", Protocol Specification, Verification and Testing IX, IFIP WG6.1 workshop, Twente, June 1989.

[Piatk86] Piatkowski T. F., "The State of the Art in Protocol Engineering", ACM SIGCOMM'86, pp. 13-18.

[Pneul77] Penuli Amir., "The Temporal Semantics of Concurrent Porgrams", Eighteenth Ann. Symposium Foundations of Computer Science, pp. 46-57, October 1977.

[PoSm82] Pozefsky D.P., and Smith F.D., "A Meta-Implementation for Systems Network Architecture", IEEE Transactions on Communications, COM-30, No.6, pp.1348-1355, June 1982.

[Potter85] Potter R.M., "ISDN Protocol and Architecture Models," Computer Networks and ISDN Systems 10, pp. 157-166, 1985.

[PrUr83] Probert R.L., Ural Hasan., "Requirements for a Test Specification Language for Protocol Implementation Testing," Protocol specification, Testing and Verification, IFIP 1983, pp. 437-443.

[Rath87] Rathgeb E.P. et al, "Protocol Testing for the ISDN D_Channel Network Layer," Proceedings of the 7th International Symposium on Protocol Specification, Testing and Verification, pp 421–434, May 1987.

[RoCa81] Routhorn G.A., Carruthers P.A., "Teletex and it's Protocols", Computer Message System, North-Holland Publishing Company, IFIP 1981.

[RoGo84] Rosier. L. E., and Gouda M. G., "Deciding progress for a class of communicating finite state machines, " Proc. Conference Information Sciences and Systems, Princeton Univ., 1984.

[Rose90] Rose M.T., "The Open Book: A Practical Perspective on Open systems Interconnection," Prentice Hall, 1990.

[RuWe82] Rubin J., and West C. H., "An improved protocol validation technique", Computer Networks, Vol. 6, Apr. 1982, pp. 66-73.

[SaCo89] Saqui-Sannes P., Courtiat J.P., "From the Simulation to the Verification of Estelle Specifications", in Proceedings of the 2nd International Conference on formal Description Techniques (FORTE89), Canada.

[SaDa85] Sabnani K., Dahbura A., "A Procedure for Generating Protocol Tests," Computer Comm. Review, Vol. 15, No. 4, Dec 1985.

[SaBo84] Sarikaya B., Bochmann G., "Synchronization and Specification Issues in Protocol Testing," IEEE Trans. on Communications, Vol. Com-32, No. 4, pp.389–395, April 1984.

[SaBo87] Sarikaya B., Bochmann G., Cerny E., "A Test Design Methodology for Protocol Testing," IEEE Trans. on Soft. Eng., Vol. SE-13. No. 5, 518–531, May 1987.

[SaPr89] Saleh Kassam., Probert Robert L., "A Synthetic Localized Approach to Protocol Validation," Technical Report No. TR-89-44, University Ottawa, Canada.

[SaSc82] Sabnani K., and Schwartz M., "Verification of a Multidestination Protocol Using Temporal Logic", Protocol Specification, Testing and Verification, North Holland Publishing Company, 1982, pp.21-42.

[SaSc84] Sabnani K., and Schwartz M., "Verification of a Multidestination Selective Repeat Protocol", Computer Networks, Vol.8., No.5., Dec 1984, pp.463-478.

[SeBo86] Serre J. M., Bochman G.v., S "A Methodology for implementing high level communication protocols", Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences, 1986.

[Select86] "Special Issue on ISDN", IEEE Journal on Selected Areas in Communications, Vol. SAC-4, No.3., May 1986.

[ShHo86] Sherif M. H., Hoover G., Wiederhold R.P., "X.25 Conformance Testing - A Tutorial," IEEE Communication Magazine, Vol-24, No. 1, pp. 16–27, Jan. 1986.

[SiBl90] Sidhu D. P., Blumer T.P., "Semi-automatic Implementation of OSI Protocols", Computer Networks and ISDN Systems, Jan 1990.

[SiCh89] Sidhu D.P, Chung A., "Experience with Formal Methods in Protocol Development", FORTE 89, Vancouver, Canada, Dec 1989.

[SiDa82] Simon Gerald A., Kaufman David J., "An Extended Finite State Machine Approach to Protocol Specification," Protocol specification, Testing and Verification, IFIP 1982.

[Svobo88] Svobodova L., "Implementing OSI Systems", Research Report #62131, IBM Research Division, Zurich Laboratory, 1988.

[TaSh88] Takahashi K., et al., "An Intelligent Support System for Protocol and Communication Software Development", IEEE Journal on Selected Areas in communications, Vol. 6, No.5, June 1988.

[TaVa86] Tanenbaum Andrew S., Van Renesse Robbert., "Distributed Operating Systems", Computing surveys, vol. 17, No.4, pp.419-470, Dec. 1985.

[TeBl81] Tenney R.L., Blumer T.P., "A Formal Specification Technique and Implementation Method for Protocols", ICST Report No. ICST/HLNP 81-15, National Bureau of Standards, Gaithersburg, MD.20890, July 1981.

[TeNg90] Teng Peng., NG Peter., "Supporting Service Development for Intelligent Networks," IEEE Journal Of Selected Areas in Communications, Vol. 8, No. 2, 1990.

[Tutor90] Tutorial Notes on ISDN Testing by Wing-Man Chan, BNR, at the Tenth International IFIP WG 6.1 Symposium on Protocol Specification, Testing and Verification, Ottawa, June 1990.

[UrPr84] Ural H. and Probert R.L., "Automated testing of protocol specifications and their implementations", in Proc. ACM SIGCOMM Symp., 1984.

[UyLa90] Umit Uyar et. al., "Algirithmic Verification of ISDN Network Layer Protocol", AT&T Technical Journal, Jan/Feb 1990.

[VePi86] Venkatraman R.C., Piatkowski Thomas F., "A Formal Comparison of Formal Protocol Specification Techniques", Protocol Specification, Testing and Verification V, IFIP 1986.

[ViLo86] Vissers. Chris A., Logrippo. Luigi., "The importance of the service concept in the design of data communications protocols", IFIP 1986.

[VoLa88] Vong. Son T., Lau A. C., "Semiautomatic implementation of protocols using an Estelle C Compiler", IEEE Transactions on Software Engineering, Vol. 14, No.3, March 1988.

[West86] West C. H., "Protocol Validation by Random State Exploration," Proceedings of the Sxith International Workshop on Protocol Specification, Testing and Verification, Montreal, Canada, 1986.

[WeZa78] West C. H., and Zafiropulo P., "Automated validation of a communications protocol: The CCITT X.21 recommendations," IBM Journal Res. Development., Vol. 22, Jan 1978.

[Zafi80] Zafiropulo P., et al, "Towards Analyzing and Synthesizing Protocols," IEEE Trans. on Communications, Vol. Com-28, No. 4, pp. 651–660, April 1980.

[Zafi83] Zafiropulo P., et al., "Protocol Analysis and Synthesis using a State Transition Model," Computer Network Architectures and Protocols, P. E. Green (Ed.), Plenum Press, May 1983, pp.645-670.

[Zimm80] Zimmermann H., "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection, IEEE Trans. Comm. Vol.COM-28, pp.425-432, April 1980.

Appendix A.1

Simplified Q.931 Protocol specification in Estelle - Level_1

```
specification ISDN systemprocess;
                    { This is the top level module for entire ISDN Specification  }
default individual queue;          { Default Don't share queue interactions       }
timescale seconds;                 { Timeouts are in seconds                      }

const
      MAX_MESSAGE_SIZE  = 256;          {Max ISDN message size is 256 bytes }
      BASIC_B_CHANNELS  = 2;            {Number of B channels for Basic Access}
      HL_SETUP_REQ      = 7D65;         {Just an example value in HEX        }
                        {Similarly assign values for other Primitive Names  }
      REQUEST           = 1;            {Prim Type variable = 1 means it is a REQ}
      RESPONSE          = 4;  .....;
type
      UserCEPId         = BASIC_B_CHANNELS;
      OctetType         = Char;

      User_Data_Type    = record
                              array[1..MAX_MESSAGE_SIZE] of OctetType;
                          end;
      PrimitiveID_Type  = {NL_SETUP_REQ, NL_CLEAR_REQ, DL_DATA_REQ....};
                              {All the possible primitives are listed above}
      PrimitiveType_Type = {REQUEST, CONFIRM, INDICATION, RESPONSE};
                              {All the possible primitive types are listed above}
      UserPrimType      = User_Data_Type;
      DL_DataType       = User_Data_Type;
```

```
                { The primitive structure may look like the ones in this box in a
real                                    environment}

UserPrimType        = record
                          ....
                          Primitive_ID  : PrimitiveID_Type;
                                {Eg., HL_Setup_Req = 7D65 (In hex) }
                          Prim_type     : PrimitiveType_Type;
                          {Request, Confirm, Response or Indication Types}
                          Data_Present  : Boolean;
```

```
                                    {If true, data is accompanying the primitive}
                            Data_Rec :  User_Data_Type;
                            {Access and interpret the data from above array}
                            Return_Code : Return_Code_Type;
                            {If applicable read check the return code}
                            ....
                            {There may be more parameters, depends on design}
                    end;

DL_DataType         = record
                            Primitive_ID : Primitive_ID_Type;
                            Prim_type : Primitive_type_type;
                            Data_Present : Boolean;
                            Data_Ptr :  User_Data_Type;
                            Return_Code : Return_Code_Type;
                                ....
                            {There may be more parameters, depends on design}

                    end;
```

```
{*************************************************************************}
{CHANNEL DEFINITION SECTION                                               }
{*************************************************************************}

    {*********************************************************************}
    {Channel between UserofQ931 and Q931AsProvider                        }
    {*********************************************************************}
    channel CHAN_Q931Access(ROLE_UserOfQ931, ROLE_Q931AsProvider)
    by ROLE_UserOfQ931:
        NL_SetupReq(NL_PrimitiveReq : UserPrimType);
            {User of Q931 sends a request to Q931 to send a setup}

        NL_CallProcReq(NL_PrimitiveReq : UserPrimType);
            {User of Q931 sends a request for alerting to be sent to the peer}

        NL_AlertingReq(NL_PrimitiveReq : UserPrimType);
            {User of Q931 sends a request for alerting to be sent to the peer}

        NL_ClearReq(NL_PrimitiveReq : UserPrimType);
            {User of Q931 sends a request to tear down the call}

        NL_ConnectReq(NL_PrimitiveReq : UserPrimType);
            {User of Q931 sends a request for connect to be sent to peer}
        NL_GenericRsp(NL_PrimitiveCnf : UserPrimType);
            {Generic Response for all the incoming indications}


    by ROLE_Q931AsProvider:
        NL_SetupInd(NL_PrimitiveInd : UserPrimType);
```

```
        NL_AlertInd(NL_PrimitiveInd  :  UserPrimType);
        NL_CallProcInd(NL_PrimitiveInd  :  UserPrimType);
        NL_ConnectInd(NL_PrimitiveInd  :  UserPrimType);
        NL_ClearInd(NL_PrimitiveInd  :  UserPrimType);
        NL_GenericCnf(NL_PrimitiveCnf   :  UserPrimType);
                    {Generic Confirm for all the above User Requests}

{*************************************************************************}
{Channel between ROLE_Q931AsUser and Data_Link_Layer_as_Provider }
{*************************************************************************}
        {Definition of Data Link Service Primitives}
channel CHAN_LAPDAccess(ROLE_Q931AsUser, ROLE_LAPDAsProvider);
by ROLE_Q931AsUser :
        DL_EstablishReq(Q931_data : DL_DataType);
                {From I 441 or Q.921, request to setup TEI value with N/W }
        DL_EstablishRsp(Q931_data : DL_DataType);
        DL_DataReq(Q931_data : DL_DataType);
        DL_DataRsp(Q931_data : DL_DataType);


by ROLE_LAPDAsProvider :
        DL_EstablishInd(LAPD_data:DL_DataType);
                {From I.441, request from N/W to setup TEI value }
        DL_EstablishCnf(LAPD_data:DL_DataType);
        DL_DataCnf(LAPD_data:DL_DataType);
        DL_DataInd(LAPD_data:DL_DataType);



{************************************************}
{MODULE HEADER DEFINITIONS SECTION       }
{************************************************}


        {*********************************************}
        {Definition of the Higher_Layer Enity        }
        {*********************************************}
        module MOD_UserOfQ931_Type  process
                ip U : array [UserCepId] of
                        CHAN_Q931Access (ROLE_UserOfQ931)
                                                common    queue;
                                {UserCepId is the number of channels for basic, therefore }
                                {       there will be as many users                       }
                end;


        {*************************************}
        {Definition of the Q931 Entity        }
        {*************************************}
        module MOD_Q931Entity_Type  process

                ip U  : array[UserCepId] of
                                CHAN_Q931Access(ROLE_Q931AsProvider);
                                                common    queue;
```

```
            P   : CHAN_LAPDAccess(ROLE_Q931AsUser)
                                                    individual queue;
        end;

    {*****************************************}
    {Definition of the Data_Link_Control Module}
    {*****************************************}
    module MOD_LAPDAsProvider_Type  process;
        ip P : CHAN_LAPDAccess(ROLE_LAPDAsProvider)
                                                    individual queue;
        end;


{*****************************************}
{BODY DEFINITIONS FOR MODULES      }
{*****************************************}


    {*************************************}
    {Body for Higher_Layer Enity          }
    {*************************************}
        Empty.....;

    {*************************************}
    {Body for Q931_Entity_type            }
    {*************************************}
        Empty.....;


    {*************************************}
    {Data_Link_Control Module             }
    {*************************************}
        Empty.....;

{*****************************************************************}
{Module Variable Declaration part of the specification }
{*****************************************************************}
    modvar
        Empty.....;

    initialize {initialization part of the ISDN specification}
    begin
        Empty.....;
    end;

end;  {end of ISDN specification}
```

Appendix A.2

Simplified Q931 Protocol specification in Estelle - Level 3

```
specification ISDN systemprocess;
                    { This is the top level module for entire ISDN Specification  }
default individual queue;       { Default Don't share queue interactions          }
timescale seconds;              { Timeouts are in seconds                         }

const
      MAX_MESSAGE_SIZE  = 256;        {Max ISDN message size is 256 bytes }
      BASIC_B_CHANNELS  = 2;          {Number of B channels for Basic Access}
      HL_SETUP_REQ      = 7D65;       {Just an example value in HEX         }
                        {Similarly assign values for other Primitive Names  }
      REQUEST           = 1;          {Prim Type variable = 1 means it is a REQ}
      RESPONSE          = 4;  .....;
type
      UserCEPId         = BASIC_B_CHANNELS;
      OctetType         = Char;

      User_Data_Type    = record
                              array[1..MAX_MESSAGE_SIZE] of OctetType;
                          end;
      PrimitiveID_Type  = (NL_SETUP_REQ, NL_CLEAR_REQ, DL_DATA_REQ....);
                              {All the possible primitives are listed above}
      PrimitiveType_Type = (REQUEST, CONFIRM, INDICATION, RESPONSE);
                              {All the possible primitive types are listed above}
      UserPrimType      = User_Data_Type;
      DL_DataType       = User_Data_Type;


{*************************************************************************}
{CHANNEL DEFINITION SECTION                                               }
{*************************************************************************}

      {*************************************************************************}
      {Channel between UserofQ931 and Q931AsProvider                          }
      {*************************************************************************}
      channel CHAN_Q931Access(ROLE_UserOfQ931, ROLE_Q931AsProvider)
```

```
by ROLE_UserOfQ931:
        NL_SetupReq(NL_PrimitiveReq  : UserPrimType);
                {User of Q931 sends a request to Q931 to send a setup}

        NL_CallProcReq(NL_PrimitiveReq  : UserPrimType);
                {User of Q931 sends a request for alerting to be sent to the peer}

        NL_AlertingReq(NL_PrimitiveReq  : UserPrimType);
                {User of Q931 sends a request for alerting to be sent to the peer}

        NL_ClearReq(NL_PrimitiveReq  : UserPrimType);
                {User of Q931 sends a request to tear down the call}

        NL_ConnectReq(NL_PrimitiveReq  : UserPrimType);
                {User of Q931 sends a request for connect to be sent to peer}
        NL_GenericRsp(NL_PrimitiveCnf   : UserPrimType);
                {Generic Response  for all the incoming indications}


by ROLE_Q931AsProvider:
        NL_SetupInd(NL_PrimitiveInd  : UserPrimType);
        NL_AlertInd(NL_PrimitiveInd  : UserPrimType);
        NL_CallProcInd(NL_PrimitiveInd  : UserPrimType);
        NL_ConnectInd(NL_PrimitiveInd  : UserPrimType);
        NL_ClearInd(NL_PrimitiveInd  : UserPrimType);
        NL_GenericCnf(NL_PrimitiveCnf   : UserPrimType);
                        {Generic Confirm for all the above User Requests}

{*************************************************************************}
{Channel between ROLE_Q931AsUser and Data_Link_Layer_as_Provider }
{*************************************************************************}
        {Definition of Data Link Service Primitives}
channel CHAN_LAPDAccess(ROLE_Q931AsUser, ROLE_LAPDAsProvider);
by ROLE_Q931AsUser :
        DL_EstablishReq(Q931_data  : DL_DataType);
                {From I.441 or Q.921, request to setup TEI value with N/W }
        DL_EstablishRsp(Q931_data  : DL_DataType);
        DL_DataReq(Q931_data  : DL_DataType);
        DL_DataRsp(Q931_data  : DL_DataType);


by ROLE_LAPDAsProvider :
        DL_EstablishInd(LAPD_data:DL_DataType);
                {From I.441, request from N/W to setup TEI value }
        DL_EstablishCnf(LAPD_data:DL_DataType);
        DL_DataCnf(LAPD_data:DL_DataType);
        DL_DataInd(LAPD_data:DL_DataType);
```

```
{************************************************}
{MODULE HEADER DEFINITIONS SECTION               }
{************************************************}


      {************************************************}
      {Definition of the Higher_Layer Enity            }
      {************************************************}
      module MOD_UserOfQ931_Type  process
            ip U : array [UserCepId] of
                  CHAN_Q931Access (ROLE_UserOfQ931)
                                                common   queue;
                        {UserCepId is the number of channels for basic, therefore }
                        {       there will be as many users                       }
            end;


      {************************************}
      {Definition of the Q931 Entity       }
      {************************************}
      module MOD_Q931Entity_Type  process

            ip U  : array[UserCepId] of
                              CHAN_Q931Access(ROLE_Q931AsProvider);
                                                common   queue;
              P   : CHAN_LAPDAccess(ROLE_Q931AsUser)
                                                individual queue;
            end;

      {************************************************}
      {Definition of the Data_Link_Control Module}
      {************************************************}
      module MOD_LAPDAsProvider_Type  process;
            ip P : CHAN_LAPDAccess(ROLE_LAPDAsProvider)
                                                individual queue;
            end;


{****************************************}
{BODY DEFINITIONS FOR MODULES            }
{****************************************}

      {****************************************}
      {Body  for  MOD_UserOfQ931_Type          }
      {****************************************}

      body MOD_UserOfQ931_Body  for MOD_UserOfQ931_Type;
                                                external;

      {****************************************}
      {Body  for  MOD_LAPDAsProvider_Type}
      {****************************************}
```

```
body MOD_LAPDAsProvider_Body for  MOD_LAPDAsProvider_Type;
                                                  external;

{*****************************************}
{Body  for  MOD_Q931Entity_Type       }
{*****************************************}
body Q931_Entity_body  for  MOD_Q931Entity_Type;
const
{*************************************************************** *******}
{Most of Q931 Message Related constants which are not shared with }
{       other processes should come here                          }
{*************************************************************************}

        {*************************************************************************}
        {1. Timers specified in this simplified Q931 specification are       }
        {      Values of timers are in seconds                                }
        {*************************************************************************}
        T301   = 180;                          {Alert received              }
        T303   = 4;                            {Setup sent                  }
        T305   = 15;                           {Disconnect sent             }
        T308   = 4;                            {Release sent                }
        T310   = 10;                           {Call_Proceeding sent        }
        T313   = 4;                            {Connect sent                }

        {*************************************************************************}
        {2. Peer-to-peer Message Units used in this spec are (values in Hex) }
        {*************************************************************************}
        ALERTING            = 01;
        CALL_PROCEEDING     = 02;
        CONNECT             = 07;
        CONNECT_ACK         = 0F;
        DISCONNECT          = 45;
        RELEASE             = 4D;
        RELEASE_COMPLETE    = 5A;
        SETUP               = 05;
        STATUS              = 7D;
        STATUS_ENQ          = 75;

        {*************************************************************************}
        { 3. Cause Codes used in this Specification (Values in Hex)          }
        {*************************************************************************}
        NORMAL_CALL_CLEARING    = 10;
        RESPONSE_TO_STATUS_ENQ  = 9E;
        INVALID_CALL_REFERENCE  = D1;
        MANDATORY_IE_MISSING    = E0;
        MANDATORY_IE_ERROR      = E4;
        MESSAGE_NON_EXISTANT    = E2;
        MESSAGE_NOT_IMPLEMENTED = E1;
        MESSAGE_NOT_COMPATIBLE  = E3;

        {*************************************************************************}
        { 4. Other Constants                                                 }
        {*************************************************************************}
        MAX_IE_LENGTH       = ...;
```

```
        MIN_MESSAGE_LENGTH = 4;


type
{*************************************************************}
{Types local to Q931 such as control blocks for Call reference,      }
{       B channels, Information Elements, Entire Message Structures}
{       etc.,                                                        }
{*************************************************************}
        PeerMessageType     = (ALERTING, CALL_PROCEEDING, CONNECT,
                               CONNECT_ACK, DISCONNECT, RELEASE,
                               RELEASE_COMPLETE, STATUS, STATUS_ENQ,
                               SETUP);
        ValidationType      = (MANDATORY_IE_MISSING, MANDATORY_IE_ERROR,
                               OPTIONAL_IE_IMISSING, OPTIONAL_IE_ERROR,....);
        StateType           = (U0, U1, U3, U4, U7, U9, U10, U11, U19);
        MessageHeaderType   = record
                                CallReferenceLength  = OctetType;
                                CallReferenceValue   = OctetType;
                                MessageType          = PeerMessageType
                              end;
        DL_DataType  = record array[1..MAX_MESSAGE_SIZE] of OctetType end;
        NL_DataType  = record array[1..MAX_MESSAGE_SIZE] of OctetType end;
        ANY_DataType = record array[1..MAX_MESSAGE_SIZE] of OctetType end;
        PeerDataType = record array[1..MAX_MESSAGE_SIZE] of OctetType end;
        MessageStructureType = record
                                MessageHeader = MessageHeaderType;
                                PeerData      = PeerDataType;
                              end;
        IeStructureType     = record array[1..MAX_IE_LENGTH] of OctetType end;
        CollectiveType      = ...;
                {Type declarations for all the different messages and IE's of  ISDN Q931 }
                {   CCITT Q.931 Chapter IV                                               }
                {    -ASN.1 types can be effectively used                                }
        ParseResultType     = (ACCEPT, IGNORE, REJECT...);
                {A parser decodes the received message and checks for syntax of PDU      }
                { If syntax is acceptable, then the ParseResult returned by parser is    }
                {   ACCEPT, if there is major error which cannot be ignored then it is   }
                {  REJECT etc.,                                                          }
        CauseCodeType       = ...;
                {Type declaration for Causecode from CCITT Q.931 Chapter IV              }
        Buffer_type = ...;
        CR_Block_Type = record
                        {containes call related information such as call reference,
                                B channel allocated, initiator of the call etc.,}
                      end;

var
        MessageHeader                : MessageHeaderType;
        DL_DataIndPtr, DL_DataReqPtr : DL_DataType;
        NL_DataIndPtr, NL_DataReqPtr : NL_DataType;
        MessageStructure             : MessageStructureType;
        LMessageStructure            : MessageStructureType;
        UMessageStructure            : MessageStructureType;
```

```
        CauseCodeIe                 : CauseCodeType;
        Send_buffer, Recv_buffer    : Buffer_type;
        CallReferenceBlock          : CR_Block_Type;

state U0,                                   {NULL State                 }
      U1,                                   {Call Initiated State       }
      U3,                                   {Outgoing Call Proceeding   }
      U4,                                   {Call Delivered             }
      U7,                                   {Call Received              }
      U8,                                   {Connect Request            }
      U9,                                   {Incoming Call Proceeding   }
      U10,                                  {Active state               }
      U11,                                  {Disconnecting State        }
      U19;                                  {Releasing state            }


{*************************************************************************}
{Functions required for Q931 Specification to be comprehensible           }
{*************************************************************************}
      {*******************************************************************}
      {Name : Start Timer                                                 }
      {*******************************************************************}
      {Function:    The calling routine passes in the ID of the timer to be }
      {             started. This procedure in turn will invoke the system  }
      {             routine or if this is the standard interface then the timer }
      {             will be automatically started for the duration sent. Once the}
      {             duration expires, a timer expired interrupt will be sent to }
      {             Q931 specification.                                     }
      {Inputs:      1.Timer Id and Duration                                 }
      {*******************************************************************}
      procedure   StartTimer(TimerID : TimerIDType;
                             Duration : DurationType);primitive;
      {Above Timer related Variables and Types are not declared and not defined}

      {*******************************************************************}
      {Name : Compatible                                                  }
      {*******************************************************************}
      {Function:    A table of compatible messages should be prepared from  }
      {             ISDN Q.931. For example once a DISCONNECT message is    }
      {             receive then Q.931 can expect a DISCONNECT, RELEASE or  }
      {             RELEASE COMPLETE only. Reception of any other message is }
      {             in-compatible. Similar information should be gathered   }
      {             for every Q.931 state.                                  }
      {Inputs:      1.Present state of Q.931 and incoming message           }
      {outputs      2.Function returns compatible or incompatible           }
      {*******************************************************************}
      function          Compatible(in_state : StateType;
                                   in_message : PeerMessageType) :
                                                        boolean;
                                                              primitive;
      {*******************************************************************}
      {Name : BuildState                                                  }
      {*******************************************************************}
      {Function:    This procedure will build the Information Element which }
```

```
{              consists of the present state. This is sent in order for the   }
{              network to determine the compatibility of its present state    }
{Inputs:       1.the Call Control Block                                       }
{outputs       2.The formatted IE is stored in IeStructure whose pointer is   }
{                 is returned.                                                }
{**************************************************************************}
procedure   BuildState(CallReferenceBlock:CR_Block_Type;
                       var IeStructure : IeStructureType);
                                                              primitive;

{**************************************************************************}
{Name : SetupCallReferenceBlock                                            }
{**************************************************************************}
{Function:     Every ISDN call related parameter such as call reference    }
{              value, B channel used, state of the call will be controlled }
{              by the block which is returned by this procedure            }
{              This procedure will make system related calls to acquire the}
{              CallReferenceBlock of appropriate size, will initialize the }
{              block and return the Block pointer to the caller            }
{Inputs:       1.The array pointer sent by data link layer or User of Q931 }
{outputs       2.Pointer to initialized area of CallReferenceBlock         }
{**************************************************************************}
procedure   SetupCallReferenceBlock(DL_DataReqPtr: ANY_DataType;
                          var CallReferenceBlock:CR_Block_Type);
                                                              primitive;

{**************************************************************************}
{Name : FormatLowerPrimitive                                               }
{**************************************************************************}
{Function:     The Peer Message formatted by the procedure                 }
{              FormatLowerPrimitive is actually sent to link control layer }
{              packed in a primitive by this procedure.                    }
{Inputs:       1.The MessageStructure record, where in the                 }
{                    formatted peer message exists                         }
{              2.The Record for data link primitive                        }
{**************************************************************************}
procedure   FormatLowerPrimitive(var DL_DataReqPtr: DL_DataType;
                    MessageStructure : MessageStructureType);
                                                              primitive;
{**************************************************************************}
{Name : FormatHigherPrimitive                                              }
{**************************************************************************}
{Function:     The important information which the user of Q931 requires   }
{              from the peer message received is packed into the User Data }
{              Request record and sent out.                                }
{Inputs:       1.The MessageStructure record, where in the                 }
{                    formatted peer message exists                         }
{              2.The Record for Higher Layer primitive                     }
{**************************************************************************}
procedure   FormatHigherPrimitive(var NL_DataReqPtr: NL_DataType;
                    MessageStructure : MessageStructureType);
                                                              primitive;
```

```
{*************************************************************************}
{Name : FormatPeerMessage                                                 }
{*************************************************************************}
{Function:    On reception of a peer message or primitive from the        }
{             adjacent layers, if the Q931 layer decided to send a        }
{             Message, the Message should be formmated making use of      }
{             Message Syntax, before the message is sent to the data link }
{             layer and eventually to the peer                            }
{Inputs:      1.Message Type which has to be formatted                    }
{             2.Message Structure Record which contains the formatted     }
{                   message                                               }
{             3.Any Ie structure if it has to be included in the message  }
{*************************************************************************}
procedure   FormatPeerMessage(MessageType : PeerMessageType;
                  var MessageStructure : MessageStructureType;
                  IeStructure : IeStructureType);              primitive;

{*************************************************************************}
{Name : ParseMessagePreamble                                              }
{*************************************************************************}
{Function:    When a peer message is received, this functions parses      }
{             the preamble and ensures the format is O.K before           }
{             a transition takes place                                    }
{Inputs:      1.Message Header contains the Message Preamble Information}
{                   and it is returned to the caller                      }
{             2.The peer message reaches Q931 as a DataInd from data      }
{                   link layer, therefore DL_DataInd record is required   }
{*************************************************************************}
function
      ParseMessagePreamble(DL_DataIndPtr :     DL_DataType;
                           Var MessageStructure : MessageStructureType)
                                                          : boolean;
                                                          Primitive;

   begin
      MessageStructure.MessageHeader.CallReferenceValue  := 0;
      MessageStructure.MessageHeader.CallReferenceLength  :=0;
      return_code    := FALSE;
            {Ensure peer message length !< MIN_MESSAGE_LENGTH      }
            {Check to ensure Protocol Discriminator is Valid       }
            {Ensure Call Reference Length and Flag are Valid else  }
            {      return_code := INVALID_CALL_REFERENCE           }
            {If the above checks are valid then return_code := TRUE }
        if (return_code != FALSE) then
          begin
            MessageStructure.MessageHeader.CallReferenceLength        :=
                                                Copy from DL_DataInd..
                                    {logical bit operations may be required}
            MessageStructure.MessageHeader.CallReferenceValue         :=
                                                Copy from DL_DataInd...
                                    {logical bit operations may be required}
            MessageStructure.MessageHeader.MessageType                :=
                                                Copy from DL_DataInd...
```

```
                                    {logical bit operations may be required}
        end;
      else
                                    {ignore_message;                        }
      ParseMessagePreamble := return_code;
  end;
{*****************************************************************************}
{Name : BuildCauseCode                                                        }
{*****************************************************************************}
{Function:    This procedure will Format the CauseCode IE which will be       }
{             a part of message to be sent to the peer                        }
{*****************************************************************************}
procedure BuildCause(CauseCode : Char;
                     var CauseCodePtr : IeStructureType); primitive;

{*****************************************************************************}
{Name : ParsePeerMessage                                                      }
{*****************************************************************************}
{Function:    This procedure will parse the incoming layer 2 DL_DataInd       }
{             which contains the peer message except the Preamble and         }
{             saves it in a global structure                                  }
{*****************************************************************************}
procedure ParsePeerMessage(MessageToParse : PeerMessageType;
                  var MessageStructure : CollectiveType;
                  var ParseResult      : ParseResultType);   primitive;

{*****************************************************************************}
{Name : ValidateMessage                                                       }
{*****************************************************************************}
{Function:    This procedure will Validate the received peer message          }
{             after it is parsed                                              }
{*****************************************************************************}
function ValidateMessage(MessageToValidate : PeerMessageType;
                         MessageHeader : MessageHeaderType) :
                               ValidationType;              primitive;

  var
      ValidationResult   : ValidationType;
      MessageToParse     : PeerMessageType;
      ParseResult        : ParseResultType;
      MessageStructure   : CollectiveType;

  begin
      {Initialize variables                                                   }
      ParseResult = ACCEPT;
      MessageToParse     := MessageToValidate;
      ParsePeerMessage(MessageToParse, MessageStrcture, &ParseResult);
          {depending upon the ParseResult and the message stored              }
          {in the MessageStrcture the programmer should validate              }
          {the Message (if ParseResult = ACCEPT) then return the              }
          {validation result to the caller                                    }
      ValidationResult   := ...;
      ValidationMessage  := ValidationResult;
  end;
```

```
initialize                          { initialization Q931 state}

to U0N0                             { the initial ISDN state is U0N0 }
  begin                             { initialize variables required for
                                        transition                    }
      Call_Reference = 0;
     T301_Timer     = T301;
     T303_Timer     = ...;
     T305_Timer     = ...;
     T308_Timer     = ...;
     T312_Timer     = ...;
      ..............
  end;
{*************************************************************}
{transition-declaration-part of the Q931_Entity process}
{*************************************************************}

trans
   from U0
       when P.DL_DataInd
          to same
              provided ((ParseMessagePreamble(DL_DataIndPtr,MessageStructure) =
                        TRUE) and (MessageStrcuture.MessageHeader.MessageType
                         = SETUP) and (ValidateMessage(SETUP,MessageStructure) =
                        MANDATORY_IE_MISSING) )
                    begin
                        BuildCause(MANDATORY_IE_MISSING,CauseCodeIe);
                        FormatPeerMessage(RELEASE_COMPLETE,
                              LMessageStructure,CauseCodeIe);
                        FormatLowerPrimitive(DL_DataReqPtr, LMessageStructure);
                        output P.DL_DataReq(DL_DataReqPtr);
                    end;

              provided ((ParseMessagePreamble(DL_DataIndPtr, MessageStructure) =
                        TRUE) and (MessageStrcuture.MessageHeader.MessageType
                         = SETUP) and (ValidateMessage(SETUP,MessageStructure) =
                        MANDATORY_IE_ERROR))
                    begin
                        BuildCause(MANDATORY_IE_ERROR,CauseCodeIe);
                        FormatPeerMessage(RELEASE_COMPLETE,
                              LMessageStructure,CauseCodeIe);
                        FormatLowerPrimitive(DL_DataReqPtr, LMessageStructure);
                        output P.DL_DataReq(DL_DataReqPtr);
                    end;

              provided ((ParseMessagePreamble(DL_DataIndPtr, MessageStructure) =
                        TRUE) and (MessageStrcuture.MessageHeader.MessageType
                         = STATUS))
                    begin
                        BuildCause(MESSAGE_NOT_COMPATIBLE, ...);
                        FormatPeerMessage(RELEASE_COMPLETE, ...);
                        FormatLowerPrimitive(...);
```

```
                output  P.DL_DataReq(...);
            end;

        provided otherwise
            begin
                BuildCause(INVALID_CALL_REFERENCE, ...);
                FormatPeerMessage (RELEASE_COMPLETE, ...);
                FormatLowerPrimitive(...);
                output  P.DL_DataReq(...);
            end;


    to  U9                  {outgoing call proceeding state}
        provided ((ParseMessagePreamble(DL_DataIndPtr, MessageStructure) =
                TRUE) and (MessageStructure.MessageHeader.MessageType
                = SETUP) and (ValidateMessage(SETUP,..) = NO_ERROR))
            begin
                SetupCallReferenceBlock(DL_DataIndPtr,
                                        CallReferenceBlock);
                FormatPeerMessage(CALL_PROCEEDING, ...);
                FormatLowerPrimitive(...);
                FormatHigherPrimitive(NL_DataIndPtr, UMessageStructure);
                StartTimer(TimerID, T310);   { 10 sec timer }
                output U[UserCEPId].NL_SetupInd(NL_DataIndPtr);
                output   P.DL_DataReq(...);
            end;

when U[UserCEPId].NL_SetupReq
    to U 1
            begin
                SetupCallReferenceBlock(NL_SetupReq,
                                        CallReferenceBlock);
                FormatPeerMessage(SETUP, ...);
                FormatLowerPrimitive(...);
                StartTimer(.,T303);    { 4 sec timer }
                output P.DL_DataReq( ...);
                end;

     to same
         provided  otherwise
            begin
                output U[UserCEPId].NL_GenericCnf( {send
                                    RC=INVALID_REQUEST } );
            end;

from U1
    when P.DL_DataInd
        to U 1 0          {Call is connected in this state }
            provided ((ParseMessagePreamble(DL_DataIndPtr, MessageStructure) =
                TRUE) and (MessageStructure.MessageHeader.MessageType
                = CONNECT) and (ValidateMessage(CONNECT,...)
                                                = NO_ERROR))
            begin
                StopTimer(T303);
```

```
                FormatPeerMessage(CONNECT_ACK, ...);
                FormatLowerPrimitive(...);
                FormatHigherPrimitive(...);
                output U[UserCEPId].NL_ConnectInd(...);
                output P.DL_DataReq(...);
        end;

to U 1 2
    provided ((ParseMessagePreamble(DL_DataIndPtr, MessageStructure) =
                TRUE) and (MessageStrcuture.MessageHeader.MessageType
                = CONNECT) and
                (ValidateMessage(CONNECT,MessageStructure,.) =
                MANDATORY_IE_ERROR))
        begin
                FormatHigherPrimitive(NL_ClearIndPtr,UMessageStructure);
                output U[UserCEPId].NL_ClearInd(NL_ClearIndPtr);
        end;


to U 3
    provided ((ParseMessagePreamble(DL_DataIndPtr, MessageStructure) =
                TRUE) and (MessageStructure.MessageHeader.MessageType
                = CALL_PROCEEDING) and
                (ValidateMessage(CALL_PROCEEDING,..) = NO_ERROR))
        begin
                StopTimer(T303);
                StartTimer(T310);
                FormatHigherPrimitive(NL_DataIndPtr, UMessageStructure);
                output U[UserCEPId].NL_CallProcInd(...);
        end;

to U 4
    provided ((ParseMessagePreamble(DL_DataIndPtr, MessageStructure) =
                TRUE) and (MessageStructure.MessageHeader.MessageType
                = ALERT) and (ValidateMessage(ALERT,..) = NO_ERROR))
        begin
                StopTimer(T303);
                FormatHigherPrimitive(NL_DataIndPtr, UMessageStructure);
                output U[UserCEPId].NL_AlertingInd(...);
        end;

to U 1 2
    provided ((ParseMessagePreamble(DL_DataIndPtr, MessageStructure) =
                TRUE) and (MessageStructure.MessageHeader.MessageType
                = DISCONNECT) and (ValidateMessage(DISCONNECT,..) =
                NO_ERROR))
        begin
                FormatHigherPrimitive(NL_ClearIndPtr,
                                        UMessageStructure);
                output U[UserCEPId].NL_ClearInd(...);
        end;

    provided ((ParseMessagePreamble(DL_DataIndPtr, MessageStructure) =
                TRUE) and (MessageStructure.MessageHeader.MessageType
```

```
                = RELEASE) and (ValidateMessage(RELEASE,..) =
                                                    NO_ERROR))
        begin
              StopTimer(T303);
              FormatPeerMessage(RELEASE_COMPLETE, ...);
              FormatLowerPrimitive(...);
              FormatHigherPrimitive(...);
              output U[UserCEPId].NL_ClearInd(...);
              output  P.DL_DataReq(...);
        end;

to U 0
    provided  ((ParseMessagePreamble(NL_DataIndPtr, MessageStructure) =
              TRUE) and (MessageStructure.MessageHeader.MessageType
               = RELEASE_COMPLETE) and
              (ValidateMessage(RELEASE_COMPLETE,..) = NO_ERROR))
        begin
              StopTimer(T303);
              output U[UserCEPId].NL_ClearInd(...);
        end;

 to same
    provided  ((ParseMessagePreamble(DL_DataIndPtr, MessageStructure) =
              TRUE) and (Compatible(Send_Present_State,
                 MessageStructure.MessageHeader.MessageType)
                                                = FALSE))

       begin
         BuildCause(MESSAGE_NOT_COMPATIBLE, ...);
         BuildState(...);
         FormatPeerMessage(STATUS, ...);
         FormatLowerPrimitive(...);
         output  P.DL_DataReq(...);
      end;

    provided (Message_Header.Message_Type  =  STATUS_ENQUIRY)
       begin
         Build_Cause_State(Cause_Code  =  81);
         FormatPeerMessage(STATUS, ...);
         FormatLowerPrimitive(...);
         output  P.DL_DataReq(...);
      end;

     provided otherwise
                        { Received a message other than above ones}
       begin
         Build_Cause_State(cause_code  =  98)
         FormatPeerMessage(STATUS, ...);
         FormatLowerPrimitive(...);
         output  P.DL_DataReq(...);
      end;

to  same
```

```
    provided ((ParseMessagePreamble(DL_DataIndPtr, MessageStructure) =
                TRUE) and (Compatible(Send_Present_State,
                  MessageStructure.MessageHeader.MessageType)
                  = TRUE) and (ValidateMessage(Incoming_Message,
                      ..) = MANDATORY_IE_MISSING) )
          begin
                BuildCause(MANDATORY_IE_MISSING,IeStructure,...);
                BuildState(CallReferenceBlock, IeStructure, ...);
                   FormatPeerMessage(STATUS,LMessageStructure,IeStructure);
                FormatLowerPrimitive(...);
                output P.DL_DataReq(...);
          end;

to U 0
    provided ((ParseMessagePreamble(DL_DataIndPtr, MessageStructure) =
                TRUE) and (MessageStrcuture.MessageHeader.MessageType
                 = RELEASE) and
                 (ValidateMessage(RELEASE,MessageStructure,..) =
                MANDATORY_IE_MISSING))
          begin
                BuildCause(MANDATORY_IE_MISSING,IeStructure);
                FormatPeerMessage(RELEASE_COMPLETE,
                       LMessageStructure,IeStructure);
                FormatLowerPrimitive(DL_DataReqPtr, I MessageStructure);
                output P.DL_DataReq(DL_DataReqPtr);
          end;

    provided ((ParseMessagePreamble(DL_DataIndPtr, MessageStructure) =
                TRUE) and (MessageStrcuture.MessageHeader.MessageType
                 = RELEASE_COMPLETE) and
              (ValidateMessage(RELEASE_COMPLETE, LMessageStructure,..)
                = MANDATORY_IE_MISSING))
          begin
                   FormatHigherPrimitive(NL_ClearIndPtr, UMessageStructure);
                output U[UserCEPId].NL_ClearInd(...);
          end;

to U 1 2
    provided (ParseMessagePreamble(DL_DataIndPtr, MessageStructure) =
                TRUE) and (MessageStrcuture.MessageHeader.MessageType
                 = DISCONNECT) and
                 (ValidateMessage(RELEASE,MessageStructure,..) =
                MANDATORY_IE_MISSING)
          begin
                   FormatHigherPrimitive(NL_ClearIndPtr, UMessageStructure);
                output U[UserCEPId].NL_ClearInd(NL_ClearIndPtr,..);
          end;
to same
    provided ((ParseMessagePreamble(DL_DataIndPtr, MessageStructure) =
                TRUE) and (Compatible(Send_Present_State,
                  MessageStructure.MessageHeader.MessageType)
                  = TRUE) and (ValidateMessage(Incoming_Message,
                      ..) = MANDATORY_IE_ERROR) )
          begin
```

```
                        BuildCause(MANDATORY_IE_ERROR,IeStructure,...);
                        BuildState(CallReferenceBlock, IeStructure, ...);
                            FormatPeerMessage(STATUS,LMessageStructure,IeStructure);
                        FormatLowerPrimitive(...);
                        output P.DL_DataReq(...);
                    end;

        to U 0
            provided (ParseMessagePreamble(NL_DataInd, MessageStructure) =
                        TRUE) and (MessageStrcuture.MessageHeader.MessageType
                        = RELEASE) and
                        (ValidateMessage(RELEASE,MessageStructure) =
                        MANDATORY_IE_ERROR)
                    begin
                        BuildCause(MANDATORY_IE_ERROR,IeStructure);
                        FormatPeerMessage(RELEASE_COMPLETE,
                                LMessageStructure,IeStructure);
                        FormatLowerPrimitive(DL_DataReqPtr, LMessageStructure);
                        output P.DL_DataReq(DL_DataReqPtr);
                    end;

            provided (ParseMessagePreamble(DL_DataIndPtr, MessageStructure) =
                        TRUE) and (MessageStrcuture.MessageHeader.MessageType
                        = RELEASE_COMPLETE) and
                        (ValidateMessage(RELEASE_COMPLETE,MessageStructure) =
                        MANDATORY_IE_ERROR)
                    begin
                        FormatHigherPrimitive(NL_ClearInd, UMessageStructure);
                        output U[UserCEPId].NL_ClearInd(...);
                    end;

..... {Similarly continue for other States from figure 6.2 and 6.3 and CCITT Q.931}

end;    {end of transitions}

{*****************************************************************}
{Module Variable Declaration part of the specification }
{*****************************************************************}
modvar

HL_Entity       : array [UserCEPId] of MOD_UserOfQ931_Type;
Q931_Entity     : array [UserCEPId] of MOD_Q931Entity_Type;
LAPD_Entity     : MOD_LAPDAsProvider_Type;


initialize {initialization part of the ISDN specification}
  begin { module initialization}
    init LAPD_Entity with MOD_LAPDAsProvider_Body;
      all User_CEP do
        begin
          init HL_Entity[User_CEP] with MOD_UserOfQ931_Body[User_CEP];
          init Q931_Entity[User_CEP] with MOD_Q931Entity_Body[User_CEP];
          connect HL_Entity[User_CEP].U to Q931_Entity[User_CEP].U;
        end;
```

```
        connect Q931_Entity.P  to LAPD_Entity.P;
    end;


end;  {end of ISDN specification}
```

Appendix A.3

Simplified Q931 Protocol specification in Estelle - Level 4

NOTE: Many parameters and procedures are not defined completely in this implementational specification. The reason for inclusion of this appendix is only to provide a sample to introducing design constructs in a specification.

```
specification ISDN systemproc ss;
                    { This is the top level module for entire ISDN Specification  }
default individual queue;        { Default Don't share queue interactions        }
timescale seconds;               { Timeouts are in seconds                       }

const
        MAX_MESSAGE_SIZE  = 256;          {Max ISDN message size is 256 bytes }
        BASIC_B_CHANNELS  = 2;            {Number of B channels for Basic Access}
        HL_SETUP_REQ      = 7D65;         {Just an example value in HEX       }
                          {Similarly assign values for other Primitive Names  }
        REQUEST           = 1;            {Prim Type variable = 1 means it is a REQ}
        RESPONSE          = 4;  .....;
        N                 = ...;
        M                 = ...;
type
        UserCEPId         = BASIC_B_CHANNELS;
        OctetType         = Char;

        User_Data_Type    = record
                                array[1..MAX_MESSAGE_SIZE] of OctetType;
                            end;
        PrimitiveID_Type  = (NL_SETUP_REQ, NL_CLEAR_REQ, DL_DATA_REQ....);
                                {All the possible primitives are listed above}
        PrimitiveType_Type = (REQUEST, CONFIRM, INDICATION, RESPONSE);
                                {All the possible primitive types are listed above}
```

```
UserPrimType          = User_Data_Type;
DLPeerPrimType        = record
                              array[1..M] of OctetType;
                        end;
UserServPrimType      = record
                              array[1..N] of OctetType;
                        end;
DLServPrimType        = record
                              array[1..N] of OctetType;
                        end;
{Since there is no data to be sent the array need not be of MAX_MESSAGE_SIZE.
Instead, just a few octets are required to represent the type of primitive,
primitiveID, collission resolution etc.,}

DL_DataType           = User_Data_Type;


{*************************************************************************}
{CHANNEL DEFINITION SECTION                                               }
{*************************************************************************}

    {*********************************************************************}
    {Channel between UserofQ931 and Q931AsProvider                        }
    {*********************************************************************}
    channel CHAN_Q931Access(ROLE_UserOfQ931, ROLE_Q931AsProvider)
    by ROLE_UserOfQ931:
          NL_SetupReq(NL_PrimitiveReq : UserPrimType);
                {User of Q931 sends a request to Q931 to send a setup}

          NL_CallProcReq(NL_PrimitiveReq : UserPrimType);
                {User of Q931 sends a request for alerting to be sent to the peer}

          NL_AlertingReq(NL_PrimitiveReq : UserPrimType);
                {User of Q931 sends a request for alerting to be sent to the peer}

          NL_ClearReq(NL_PrimitiveReq : UserPrimType);
                {User of Q931 sends a request to tear down the call}

          NL_ConnectReq(NL_PrimitiveReq : UserPrimType);
                {User of Q931 sends a request for connect to be sent to peer}
          NL_GenericRsp(NL_PrimitiveCnf : UserPrimType);
                {Generic Response for all the incoming indications}


    by ROLE_Q931AsProvider:
          NL_SetupInd(NL_PrimitiveInd : UserPrimType);
          NL_AlertInd(NL_PrimitiveInd : UserPrimType);
          NL_CallProcInd(NL_PrimitiveInd : UserPrimType);
          NL_ConnectInd(NL_PrimitiveInd : UserPrimType);
          NL_ClearInd(NL_PrimitiveInd : UserPrimType);
          NL_GenericCnf(NL_PrimitiveCnf : UserPrimType);
          {Generic Confirm for all the above User Requests}
```

```
{*****************************************************************}
{Channel between User of Q931 and Q931 service activation Module  }
{ No primitives here result in peer-to-peer commn. directly       }
{*****************************************************************}
channel CHAN_UserServiceAccess(ROLE_UserToQ931Serv,ROLE_Q931ToUserServ)
by ROLE_UserToQ931Serv:
    NL_SYS_StartUpReq(NL_SystemServPrimitiveReq : SystemServPrimType);
            {The SystemServPrimType will be declared in the System include files }
        {User of Q931 sends a request for enabling of Q931 and lower layers-first thing}
    NL_SYS_ShutDownReq(NL_SystemServPrimitiveReq : SystemServPrimType);
        {User of Q931 sends a request for disabling of Q931 and lower layers}
    NL_ActServReq(NL_ServPrimitiveReq : UserServPrimType);
    {User of Q931 sends a request for activation of Q931 and underlying services}
    NL_DeactActServReq(NL_DeactPrimitiveReq : UserServPrimType);
        {User of Q931 sends a request for de-activation of Q931 and underlying services}

 by ROLE_Q931ToUserServ:
    NL_SYS_StartUpCnf(NL_SystemServPrimitiveReq : SystemServPrimType);
        NL_SYS_ShutDownCnf(NL_SystemServPrimitiveReq : SystemServPrimType);
        NL_ActServCnf(NL_ServPrimitiveReq : UserServPrimType);
    NL_DeactActServCnf(NL_DeactPrimitiveReq : UserServPrimType);


 {*****************************************************************}
 {Channel between Q931 as user and Data_Link_Layer_as_Provider    }
 {*****************************************************************}
 {Definition of Data Link Service Primitives}
 channel CHAN_LAPDAccess(ROLE_Q931AsUser, ROLE_LAPDAsProvider);
 by ROLE_Q931AsUser :
    DL_SYS_StartUpReq(DL_SystemServPrimitiveReq : SystemServPrimType);
    {Q931 sends a request for enabling of Lower Layer Services}
    DL_SYS_ShutDownReq(DL_SystemServPrimitiveReq : SystemServPrimType);
    {Q931 sends a request for disabling of Lower layer services}
    DL_ActSapReq(DL_ServPrimitiveReq : DLServPrimType);
    DL_DeactSapReq(DL_ServPrimitiveReq : DLServPrimType);
    DL_ConnectReq(DL_PeerPrimitiveReq : DLPeerPrimType);
    DL_DiscReq(DL_PeerPrimitiveReq : DLPeerPrimType);
    DL_DataReq(DL_PeerPrimitiveReq : DLPeerPrimType);

 by ROLE_LAPDAsProvider :
    DL_SYS_StartUpCnf(DL_SystemServPrimitiveReq : SystemServPrimType);
    DL_SYS_ShutDownCnf(DL_SystemServPrimitiveReq : SystemServPrimType);
    DL_ActSapCnf(DL_ServPrimitiveReq : DLServPrimType);
    DL_DeactSapCnf(DL_ServPrimitiveReq : DLServPrimType);
    DL_ConnectCnf(DL_PeerPrimitiveReq : DLPeerPrimType);
    DL_DiscCnf(DL_PeerPrimitiveReq : DLPeerPrimType);
    DL_DataCnf(DL_PeerPrimitiveReq : DLPeerPrimType);

{*************************************************}
{MODULE HEADER DEFINITIONS SECTION                }
{*************************************************}
```

```
{*****************************************}
{Definition of the User of Q931           }
{*****************************************}
module MOD_UserOfQ931_Type  process;
  ip U  :  array[UserCepId] of CHAN_Q931Access(ROLE_UserOfQ931):
                                        common   queue;
       US : CHAN_UserServiceAccess(ROLE_UserToQ931Serv);
  end;


{*************************************}
{Definition of the Q931 Entity        }
{*************************************}

module MOD_Q931Entity_Type  process;
  ip U  :   array[UserCepId] of CHAN_Q931Access(ROLE_Q931AsProvider);
                                        common   queue;
       US :  CHAN_UserServiceAccess(ROLE_Q931ToUserServ);
       P  :  CHAN_LAPDAccess(ROLE_Q931AsUser);
                                        individual queue;
  end;

{*************************************************}
{Definition of the Data_Link_Control Module}
{*************************************************}
module MOD_LAPDAsProvider_Type  process;
  ip P  :  CHAN_LAPDAccess(ROLE_LAPDAsProvider);
                                          individual queue;
  end;


{*****************************************}
{BODY DEFINITIONS FOR MODULES     }
{*****************************************}
   body MOD_UserOfQ931_Body  for MOD_UserOfQ931_Type;
                                                external;

   body MOD_LAPDAsProvider_Body  for MOD_LAPDAsProvider_Type;
                                                external;

{*****************************************}
{Body for Q931 Entity Type                }
{*****************************************}
body MOD_Q931Entity_Body  for  MOD_Q931Entity_Type;
{*****************************************************************}
{Definitions Internal to Q931 Entity Body                         }
{*****************************************************************}

   {*****************************************************************}
   {Definition of Channels Internal to Q931 Entity Body              }
   {*****************************************************************}

   {*****************************************************************}
   {Definition of Channel between Upper Layer State Machine and the  }
```

```
{ Lower Layer State Machine                                              }
{************************************************************************}
channel CHAN_UpLayerAccess(ROLE_UpperToLower, ROLE_LowerToUpper);
  by ROLE_UpperToLower :
          INT_UpToLoClearServReq(     );
          {parameters should indicate the source, destination & identifier of the
          internal event}
                  {When the user of Q931 issues SYS_SHUTDOWN_REQ, before the
                  layers are disabled, the existing peer-to-peer Q.931 calls should
                  be cleared and data link connection should be disconnected,
                  therefore the Upper Layer State Machine will issue the above
                  Internal event to ensure it is done.}
    .....; {More internal events should come here }

  by ROLE_LowerToUpper :
          INT_LoToUpClearServCnf(     );
    .....; {More internal events should come here }


{************************************************************************}
{Definition of Channel between LowerLayer State Machine and the          }
{ Peer Q931 State Machine                                                }
{************************************************************************}
channel CHAN_LoLayerAccess(ROLE_LowerToPsm, ROLE_PsmToLower);
  by ROLE_LowerToPsm :
          INT_LoToPSMClearCallsReq(     ); {This is an internal event }
    .....; {More internal events should come here }

  by ROLE_PsmToLower :
          INT_PsmToLoClearCallsCnf(     );
    .....; {More internal events should come here }

{************************************************************************}
{Definition of Channel between Father and Son modules for better         }
{progression in the interaction between the state machines               }
{************************************************************************}
channel CHAN_FatherSonAccess(ROLE_AsFather, ROLE_AsSon);
    by ROLE_AsSon : EndOfExecution(IntPrimitiveInfo);
                        {The record for IntPrimitiveInfo indicates whether an
                        internal event is present or not. If it is present then it
                        shows information such as source, destination and
                        identifier of the internal event}
  {The father module does not have any interaction to be  sent to the son module}

{************************************************************************}
{Definition of Upper Layer State Machine Module                          }
{************************************************************************}
module MOD_UpLayerSM_Type activity;
  ip US    : CHAN_UserServiceAccess(ROLE_Q931ToUserServ);
     P     : CHAN_LAPDAccess(ROLE_Q931AsUser);
     IN1   : CHAN_UpLayerAccess(ROLE_UpperToLower);
          {This interaction point can be connected to the interaction point of Lower
          Layer State Machine, and it carries the internal events from Upper Layer
          State Machine to Lower Layer State Machine}
```

```
    IN2  : CHAN_UpLayerAccess(ROLE_LowerToUpper);
           {This interaction point can be attached to the interaction point of Lower
           Layer State Machine, and it brings the internal events from Lower Layer
           State Machine to Upper Layer State Machine}

    FS1  : CHAN_FatherSonAccess(ROLE_AsSon);
  end;
           {This module will not receive or send interactions through U IP}
body MOD_UpLayerSM_Body MOD_UpLayerSM_Type;

{*************************************************************}
{Definition Of Local Functions                                }
{*************************************************************}
   {*************************************************************}
   {Name : Format_Primitive                                      }
   {*************************************************************}
   {Function:    A primitive which doesn't have any data to be sent is }
   {             formatted using this procedure call.            }
   {                                                             }
   {Inputs:      1. Name of Primitive Etc.                       }
   {Output       1. Pointer to structure containing the primitive }
   {*************************************************************}
   procedure  Format_Primitive(PrimitiveID : PrimitiveIDType;
                    var PrimitiveStructure : PrimitiveS)tructureType;

                                                          primitive;
   {*************************************************************}
   {Name : CheckUserSentInfo                                     }
   {*************************************************************}
   {Function:    This function checks the user options sent by user of Q.931 }
   {             and returns a true if the values are acceptable else FALSE  }
   {Inputs:      1. Pointer to location containing user sent options }
   {*************************************************************}
   function    CheckUserSentInfo(PrimRecord) : Boolean;
                                                          primitive;
   {*************************************************************}
   {Name : InitializeIntPrimitiveInfo                            }
   {*************************************************************}
   {Function:    The Information such as internal event is present or not  }
   {             the name of source state machine and the name of the      }
   {             destination state machines are copied into a local structure }
   {             which can be accessed by the parent module and subsequen-}
   {             tly link with the destination module                      }
   {*************************************************************}
   InitializeIntPrimitiveInfo(Event_Present_or_Not:  StatusType;
                              From_SM : SMType;
                              To_SM : SMType
                                 var IntInfoStreuture : IntEventType);
                                                          primitive;

   ..... {More Functions}
```

```
{*****************************************************************}
{Initialize the module MOD_UpLayerSM_Type                         }
{*****************************************************************}
initialize
   {Initialize the Upper State Machine State and other variables }
end;
{*****************************************************************}
{Sample Transitions for the State Machine Interactions for Upper  }
{      Layer Interface                                            }
{*****************************************************************}
trans
 from Disabled
   {This should be a global state which comes from Global Control Block, once the
                state is changed in one module it will be globally updated}
   when US.NL_SYS_StartUpReq(PrimRecord);
     provided (CheckUserSentInfo(PrimRecord) = TRUE)
          {The user will send  information for life cycle of the calls acceptable to
          Q.931 through the PrimRecord}
       begin
          format_primitive(DL_SYS_StartUpReq, ...);
                                {Fromat primitive before sending}
          output P.DL_SYS_StartUpReq(LayerIdentifier, PrimRecord...);
          {This output statement is equivalent to system send routine shown below}
                {SYS_send_primitive(LayerIdentifier, PrimRecord, ...);
          {The lower layer will also need information for the setting up of server
                environment, therefore PrimRecord is also included}
          InitializeIntPrimitiveInfo(NO_INTERNAL_EVENT, Upper_SM,
                          None, IntInfoRecord);
          {Pack all the information regarding internal event into the record pointed
          to by IntInfoRecord. In this instance there is no Internal Event at all
          therefore you can ignore other parameters }
          output FS1.EndOfExecution(IntInfoRecord..);
       end;
     to Enabling

 from Enabling
   when P.DL_SYS_StartUpCnf(PrimRecord);
     provided (ReturnCode  = SUCCESSFUL) {Check to ensure RC is successful}
       begin
          format_primitive(NL_SYS_StartUpCnf, DataPtr, rc...);
          {Format a primitive to be sent to the user of Q.931}
          output US.NL_SYS_StartUpCnf(LayerIdentifier, rc...);
          {Send the formatted primitive through the IP US (meant for service
                          activation related interactions)}
          InitializeIntPrimitiveInfo(NO_INTERNAL_EVENT, UP_SM, LO_SM,..);
          output FS1.EndOfExecution(IntInfoRecord,...);
       end;
 to Enabled

 from Enabled
   when US.NL_SYS_ShutDownReq(PrimRecord);
          {User of Q.931 wants to close the lower layer services}
       provided ( {State of Upper Layer State Machine = Enabled })
       begin
```

```
        InitializeIntPrimitiveInfo(INT_UpToLoClearServReq, UP_SM, LO_SM,...);
        {Initialize the internal event id, source and destination information }
        output IN1.INT_UpToLoClearServReq(IntInfoRecord);
        {Send the internal event through IN1, obviously to Lower Layer SM}
        {Lower Layer SM will send an internal event to Peer SM to clear }
        output FS1.EndOfExecution(IntInfoRecord,...);
    end;
to WaitingForDisable

from WaitingForDisable
when IN2.INT_LoToUpClearServCnf(PrimRecord);
  {Received an internal event sent by Lower Layer State Machine}
    provided ( {State of Upper Layer State Machine = Enabled })
    begin
        format_primitive(DL_SYS_ShutDownReq, DataPtr, rc...);
            {Now that peer state machine and lower state machine have cleared,
            it is possible to shut down the entire provider layers now}
        output P.DL_SYS_ShutDownReq(LayerIdentifier, rc...);
        InitializeIntPrimitiveInfo(NO_INTERNAL_EVENT, UP_SM, LO_SM,...);
        output FS1.EndOfExecution(IntInfoRecord,...);
    end;
to Disabling

from Disabling
when P.DL_SYS_ShutDownCnf(PrimRecord);
  {The lower layers are completely shut down}
    provided ( {State of Upper Layer State Machine = Enabled })
    begin
        format_primitive(NL_SYS_ShutDownCnf, DataPtr, rc...);
            {Inform the user of Q.931 about the successful shutting
                                                down of lower layers}
        output US .NL_SYS_ShutDownCnf(LayerIdentifier, rc...);
        InitializeIntPrimitiveInfo(NO_INTERNAL_EVENT, UP_SM, LO_SM,...);
        output FS1.EndOfExecution(IntInfoRecord,...);
    end;

end;


{*****************************************************************}
{Definition of the Lower Layer State Machine                      }
{*****************************************************************}
module MOD_LoLayerSM_Type activity;
  ip US   : CHAN_UserServiceAccess(ROLE_Q931ToUserServ);
     P    : CHAN_LAPDAccess(ROLE_Q931AsUser);
     IN1  : CHAN_UpLayerAccess(ROLE_UpperToLower);
     IN2  : CHAN_UpLayerAccess(ROLE_LowerToUpper);
     IN3  : CHAN_LoLayerAccess(ROLE_LowerToPsm);
     IN4  : CHAN_LoLayerAccess(ROLE_PsmToLower);
     FS2  : CHAN_FatherSonAccess(ROLE_AsSon);
end;
body MOD_LoLayerSM_Body MOD_LoLayerSM_Type;
```

```
{*****************************************************************}
{Definition Of Local Functions                                    }
{*****************************************************************}
.....

{*****************************************************************}
{Initialize the module MOD_UpLayerSM_Type                         }
{*****************************************************************}
initialize
   {Initialize the Lower State Machine State and other variables }
end;
{*****************************************************************}
{Sample Transitions for the of State Machine Interactions for Upper Layer Interface}
{*****************************************************************}
trans
 from Passive
   {This should be a global state which comes from Global Control Block, once the
                state is changed in one module it will be globally updated}
   when US.NL_ActSerReq(PrimRecord):
      begin
         format_primitive(DL_ActSapReq, SAP!, ...);
         output P.DL_ActSapReq(LayerIdentifier,PrimRecord...);
         InitializeIntPrimitiveInfo(NO_INTERNAL_EVENT, From_SM, To_SM,...);
         output FS2.EndOfExecution(IntInfoRecord..);
      end;
     to Activating

 from Activating
   when P.NL_ActSapCnf(PrimRecord);
     provided ({ReturnCode  = SUCCESSFUL})  {Ensure return code is successful}
      begin
         format_primitive(DL_ConnectReq, UCepId, rc...);
         output P.DL_ConnectReq(LayerIdentifier, PCepId    ...);
         InitializeIntPrimitiveInfo(NO_INTERNAL_EVENT, From_SM, To_SM,...);
         output FS2.EndOfExecution(IntInfoRecord..);
      end;
     to Establishing


 .....   {Similarly complete the rest of the Lower Layer State Machine }

 {The following transition will indicate the behavior of the lower state machine on
 reception of the internal event sent by the Upper layer state machine which turn
 resulted from reception of request for shutting down from user of Q.931, given
 above}
 from Established
    when IN1.INT_UpToLoClearServReq
      begin
         InitializeIntPrimitiveInfo(INT_LoToPSMClearCalls, From_SM, To_SM,...);
         output IN3.INT_LoToPSMClearCallsReq(IntInfoRecord);
         output FS2.EndOfExecution(IntInfoRecord..);
      end;
     to WaitForDisconnecting
```

```
from WaitForDisconnecting
    when IN4.INT_PSMToLoClearCallsCnf
        {Peer state machine performed peer-to-peer clearing successfully}
      begin
        format_primitive(DL_Deact_Req,  ...);
        output P.DL_Deact_Req(LayerIdentifier,PrimRecord...);
               {Now the provider data link layer should perform peer-to-peer
                    deactivation}
        InitializeIntPrimitiveInfo(NO_INTERNAL_EVENT, From_SM, To_SM,...);
        output FS2.EndOfExecution(IntInfoRecord..);
      end;
    to Disconnecting

from Disconnecting
    when P.DL_Deact_Cnf
        {Layer 2 peer-to-peer deactivation is completed successfully}
      begin
        InitializeIntPrimitiveInfo(NO_INTERNAL_EVENT, From_SM, To_SM,...);
        output FS2.EndOfExecution(IntInfoRecord..);
      end;
  to Disconnected

from Disconnected                    {this is a spontaneous transition}
      begin
        format_primitive(DL_Deact_Sap_Req,  ...);
        output P.DL_Deact_Sap_Req(LayerIdentifier,PrimRecord...);
        InitializeIntPrimitiveInfo(NO_INTERNAL_EVENT, From_SM, To_SM,...);
        output FS2.EndOfExecution(IntInfoRecord..);
      end;
    to Deactivating

from Deactivating
    when P.DL_Deact_Sap_Cnf
    begin
        InitializeIntPrimitiveInfo(INT_LoToUpClearServCnf, From_SM, To_SM,...);
        output IN2.INT_LoToUpClearServCnf(IntInfoRecord);
        {Inform Upper Layer State Machine that the lower layer services are
                                    cleared}
        output FS2.EndOfExecution(IntInfoRecord..);
      end;
    to Passive

end;

{*****************************************************************}
{Definition of the Peer State Machine Module                      }
{*****************************************************************}
module MOD_PsmSM_Type activity;
  ip U      : array[UserCEPId] of CHAN_Q931Access(ROLE_Q931AsProvider);
     P      : CHAN_LAPDAccess(ROLE_Q931AsUser);
     IN3    : CHAN_LoLayerAccess(ROLE_LowerToPsm);
     IN4    : CHAN_LoLayerAccess(ROLE_PsmToLower);
     FS3    : CHAN_FatherSonAccess(ROLE_AsSon);
  end;
```

```
   body MOD_PsmSM_Body MOD_PsmSM_Type;

initialize                          { initialization Q931 state}

to U0N0                             { the initial ISDN state is U0N0 }
  begin                             { initialize variables required for
                                        transition                    }
     Call_Reference = 0;
     T301_Timer     = T301;
     T303_Timer     = ...;
     T305_Timer     = ...;
     T308_Timer     = ...;
     T312_Timer     = ...;
     ..............
  end;
{*************************************************************************************}
{transition-declaration-part of the MOD_PsmSM_Body (same as Q931_Entity process)   }
{*************************************************************************************}

trans

            {The transitions will remain the same as in Appendix A.2. Additional
            transitions are required to handle the reception of internal primitives
            from the interaction point IN3 or for sending internal primitives through
            the interaction point IN4.}

end;

{*************************************************************************}
{ This is the TRANSITION PART of MOD_Q931Entity_Type      }
{*************************************************************************}
trans
      {This is the part which will act as the control program of figure 3.6. This part will
      have access to the global static structure of the protocol where information such as
      present state of each state machine is kept. Based upon the incoming event and the
      present state of the state machines the modules of Higher Layer State Machine or
      Lower Layer State Machine or Peer Protocol State Machine are linked dynamically
      (there will be as many Peer State Machine Modules as that of the number of Users
      of MOD_Q931Entity module. Once the transition is completed by the target
      module, the target module will return the internal event from the interaction
      points FS1, FS2 or FS3 respectively(this could also be done through external
      variables). From the interaction received through FS1, FS2 and FS3 the parent
      module can again change the dynamic links if required}
{A sample dynamic linking of Q931Entity module links is shown below}
  when US.<Any Interaction>
    provided ( {Determine_Higher_Layer_State != ENABLED } )
       begin
            attach Q931_Entity.US to USM.US;
            attach Q931_Entity.P to USM.P;
       end;
  when US.<Any Interaction>
    provided ( {Determine_Higher_Layer_State = ENABLED } and
                    {Lower_Layer_State != CONNECTED } )
       begin
```

```
            attach Q931_Entity.US to LSM.US;
            attach Q931_Entity.P to LSM.P;
        end;
  when US.<Any Interaction>
    provided ( {Determine_Higher_Layer_State = ENABLED } and
                    {Lower_Layer_State = CONNECTED } )
        begin
            attach Q931_Entity[User_CEP].U to PSM[User_CEP].U;
            attach Q931_Entity.P to PSM.P;
        end;

  when FS1.EndOfExecution
    provided ( {(Internal_Primitive = Present) and (determine source and destination
                    state machines)})
        begin
            detach Q931_Entity.US   Source.US
            detach Q931_Entity.P    Source.P
            attach Q93'_Entity.US to Destination.US;
                        { Destination is destination state machine}
            attach Q931_Entity.P to Destination.P;
        end;

        ..............{Similarly continue for other interactions}

end


{*****************************************************************************}
{Module Variable Declaration part of the Module MOD_Q931_Entity_Type }
{*****************************************************************************}
modvar

HL_Entity       : array [UserCEPId] of MOD_UserOfQ931_Type;
Q931_Entity     : array [UserCEPId] of MOD_Q931Entity_Type;
LAPD_Entity     : MOD_LAPDAsProvider_Type;
PSM             : array [UserCEPId] of MOD_PsmSM_Type;
LSM             : MOD_LoLayerSM_Type;
USM             : MOD_UpLayerSM_Type;

initialize {initialization part of the ISDN specification}
  begin { module initialization}
    init LAPD_Entity with MOD_LAPDAsProvider_Body;
        all User_CEP do
            begin
                init HL_Entity[User_CEP] with MOD_UserOfQ931_Body[User_CEP];
                init Q931_Entity[User_CEP] with MOD_Q931Entity_Body[User_CEP];
                connect HL_Entity[User_CEP].U to Q931_Entity[User_CEP].U;
                attach Q931_Entity[User_CEP].U to PSM[User_CEP].U;
            end;
        connect Q931_Entity.P to LAPD_Entity.P;
        attach Q931_Entity.FS1 to USM.FS1;
        attach Q931_Entity.FS2 to LSM.FS2;
        attach Q931_Entity.FS3 to PSM.FS3;
        {The IP's IN1-IN4 may be statically linked or dynamically linked}
```

```
  end;

end;  {end of ISDN specification}
```