This book was bound by

Badminton Press
18 Half Croft, Syston, Leicester, LE7 8LD
Telephone: Leicester (0533) 602918.

SPECIFICATION AND IMPLEMENTATION OF COMPUTER NETWORK PROTOCOLS

BY

MICHAEL C. COLLETT, B.Sc.

A Master's Thesis

Submitted in partial fulfilment of the requirements

for the award of Master of Philosophy

of Loughborough University of Technology

May 1986.

Supervisor: M.C. Woodward, B.Sc. Ph.D.

Department of Computer Studies.

SPECIFICATION AND IMPLEMENTATION OF COMPUTER NETWORK PROTOCOLS

ABSTRACT

A reliable and effective computer network can only be achieved by adopting efficient and error-free communication protocols. Therefore, the protocol designer should produce an unambiguous specification meeting these requirements. Techniques for producing protocol specifications have been the subject of intense interest over the last few years. This is partly due to the advent of an international standard for networking. A variety of methods have been employed, some of which are described in detail in this thesis.

However, even when the specification has been produced there still remains the task of implementation. A particular network may be used by machines with widely varying instruction sets. The initial implementation is often rewritten into several different languages and assembly codes. Hence there is considerable duplication of effort, and discrepancies can easily arise between the software on the different machines.

This thesis begins with a detailed analysis of current practice in the field of communication protocols and protocol specification. Following this, automatic generation of protocol software is considered. The work presented here concentrates on low-level protocols. Two specification languages are presented together with the concepts used in the language designs. The first language was implemented as part of a protocol modeling system, and the second language was used as the source language for a retargetable protocol compiler.

ACKNOWLEDGEMENTS

I wish to express my gratitude to my supervisor, Dr. M.C. Woodward, for his invaluable guidance and encouragement during the period of my research and his help in the preparation of this thesis.

I would also like to thank Prof. D.J. Evans for his help and support.

Thanks also go to Mr S. Bedi, Systems Manager of the VAX computer in the Computer Studies Department at Loughborough, for his valuable assistance and to my present employers, the Cripps Computing Centre of Nottingham University, for allowing me to use their text processing facilities for the production of this thesis.

Last, but not least, I wish to acknowledge the contribution that my parents have made to this work, by their consistent support and encouragement.

CONTENTS

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

1.1. COMPUTER NETWORKS

The ability to share information, that is to communicate, has played a vital role in the development of the human race. Modern telecommunication systems have extended this ability by allowing rapid communication over long distances. As computers came to be used in an ever increasing number of areas of human activity, it seemed desirable that they should also be given this ability. Computer networks were devised to fulfill this need.

Once computers could communicate this affected the development of computing methods. The old model of a single machine serving all the needs of an organisation has been replaced by a new model where several separate, but interconnected computers, do the job. Tanenbaum(1981) defines a computer network as:

> an interconnected collection of autonomous computers.

He also discusses the related term of distributed system. While he states that there is considerable confusion in the literature, he himself defines a distributed system as

> a special case of a network, one with a high degree of cohesiveness and transparency.

Ideally, the user of a distributed system need not know that there are multiple processors; it should behave like a single processor system.

In recent years network technologies have diversified, and there are now two main categories of computer network: local and wide area networks.

1.1.1. WIDE AREA NETWORKS

The first networks connected computers over a large geographical areas using land-line, radio or satellite communications. They were characterised by low data transfer rates between computers and, since distances were large, long delays between transmission and reception of messages within the network. Such networks usually connected multiple sites within a single organisation such as a business company. Alternatively, they were used by co-

operating organisations such as Universities and research establishments. Such networks are known as long haul or wide area networks.

The topology of many wide area networks is similar to a telephone network. Selected major sites are linked together using high-speed lines to form a trunk system. The remaining sites are each linked to the trunk system via a connection to one of these selected sites. Thus each site in the network can communicate with every other site.

1.1.2. LOCAL AREA NETWORKS

About ten years ago, there was growing interest in interconnecting computers within a localised environment. This was partly due to a desire to interconnect various types of office equipment such as mini- and micro- computers, word processing systems and printers. There was also interest in taking advantage of cheaper computing based on smaller processor units.

Before this time those local networks that had existed had been miniature wide area networks. Various alternative strategies were explored, which included buses and rings. This work revealed that using the latest technologies it was possible to achieve a moderately high data rate on the communications medium and also relatively low costs. This possibility resulted in further development and the emergence of local area networks as they are today.

Local Area Networks (LANs) are generally considered to have the following features (Clark,1978).

(1) Restricted geographical area (for example, a few kilometers).

(2) Moderately high data rates (typically 1-10 Mbit/s on the communication medium).

(3) Relatively low cost communications.

(4) A wide range of attached devices.

(5) Ownership of the LAN by a single organisation.

There are three main types of local area network. There are the bus, eg. Ethernet (Metcalf,1976), and the token ring (Saltzer,1979), which were both developed in the USA, and the slotted ring, eg. Cambridge Ring (Wilkes,1979), developed in the UK.

The typical use of a LAN is to link various computer hosts and user terminals in a large educational establishment or within a single site of a commercial organisation. Figure 1.1 shows that such a network might also include personal work stations with their own local processing power and that it can allow expensive resources such as high-speed printers and plotters to be shared. The LAN communication subsystem could be realised by any of the local area network types mentioned above, providing suitable hardware and software exists on the computing hosts connected to the network.

1.2. NETWORK PROTOCOLS

In the context of computer networks, the meaning of the word "protocol" is more restricted than in, say, the diplomatic context. A suitable working definition is this:

> A protocol is a set of rules designed to enable interaction between two or more communicating parties

This definition clearly has certain prerequisites: there must be at least two parties, and these parties must be linked by a communications medium. To interact the two parties will exchange messages via the communication medium. This exchange will not be arbitrary; format and meaning of each message and the sequence in which the are exchanged will be governed by a set of mutually agreed rules. This set of rules is a communications protocol.

At the highest level, users may wish to transfer files between computers, send electronic mail to colleagues in other places or access remote databases. At the lowest level, these functions must be carried out by electronic signals. There are clearly fundamental differences between communicating at these two levels. Owing to these differences, systems that provides network services are often built in several levels or layers. This

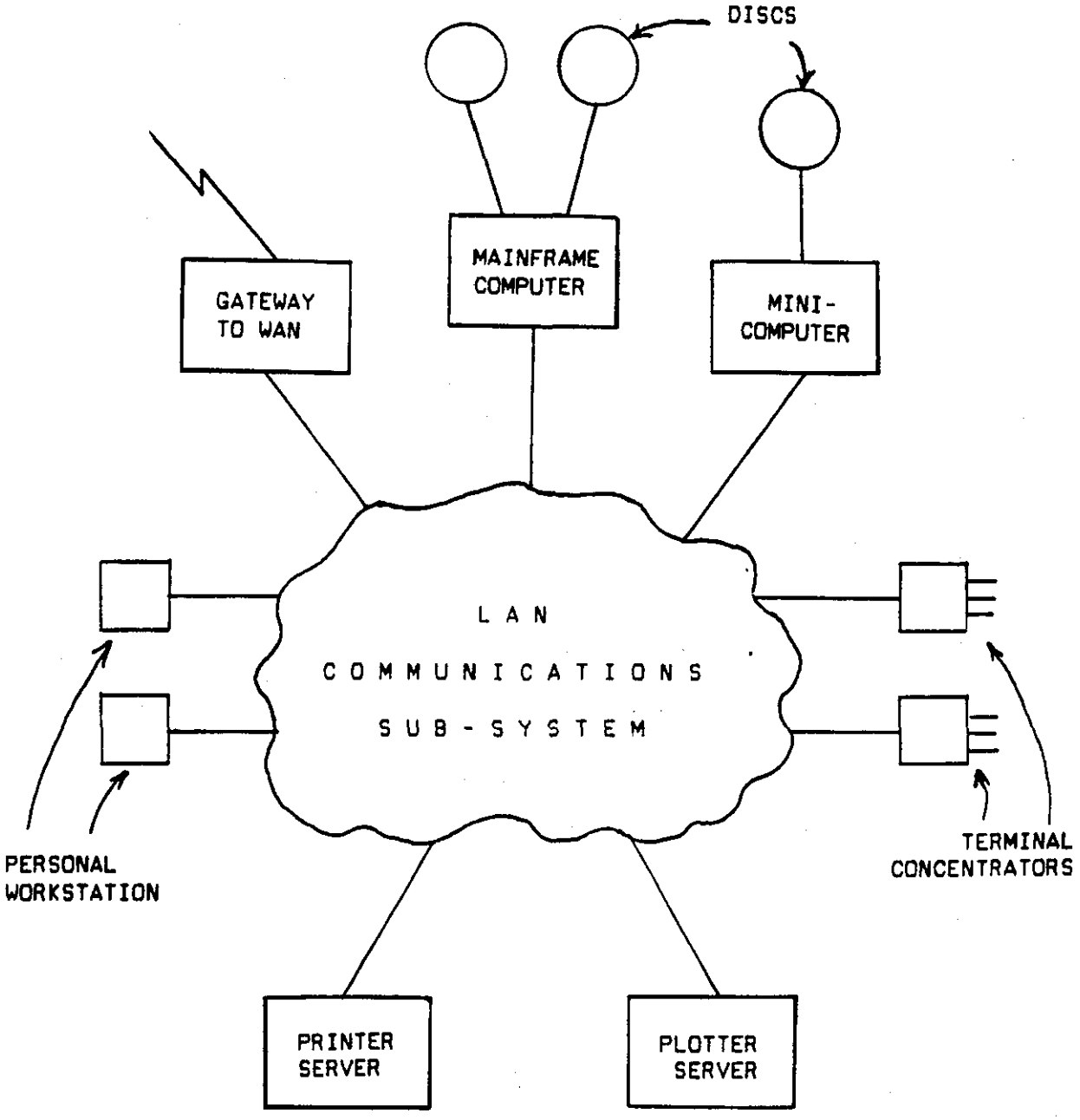


FIGURE 1.1 - A LOCAL AREA NETWORK

structured approach limits the complexity of each individual piece of protocol software, which makes design, implementation and maintenance of protocol software much easier. The general structure of a networking system is called the network architecture.

1.3. NETWORK ARCHITECTURE

A network architecture consists of layers of protocol. Each layer will have some clearly defined function. Communication within that layer is conducted between protocol entities resident on different machines. Communicating entities on different machines within a layer are known as peer entities. This is illustrated in figure 1.2. In reality no data is directly transferred between peer entities except at the lowest level. Instead, each layer passes data and control information to the layer below, until the lowest layer is reached. At the lowest level there is physical communication, as opposed to the virtual communication used by the higher layers. In figure 1.2., virtual communication is represented by dotted lines and physical communication is represented by solid lines.

Both standards organisations and computer manufacturers have produced generalised network architectures, which are known as reference models. Examples of reference models produced by standards organisation are the ISO reference model, and the IEEE 802 Standard for local area networks. An example of a reference model produced by a manufacturer is IBM's System Network Architecture (SNA).

1.4. REFERENCE MODELS

1.4.1. THE ISO REFERENCE MODEL

A reference model for Open Systems Interconnection (OSI) has been devised by the International Standards Organisation (ISO). This is described in Zimmerman(1980) and Tanenbaum(1981). It was developed as a first step towards an international standard for network architecture. Each layer is listed below together with a brief summary of its function.

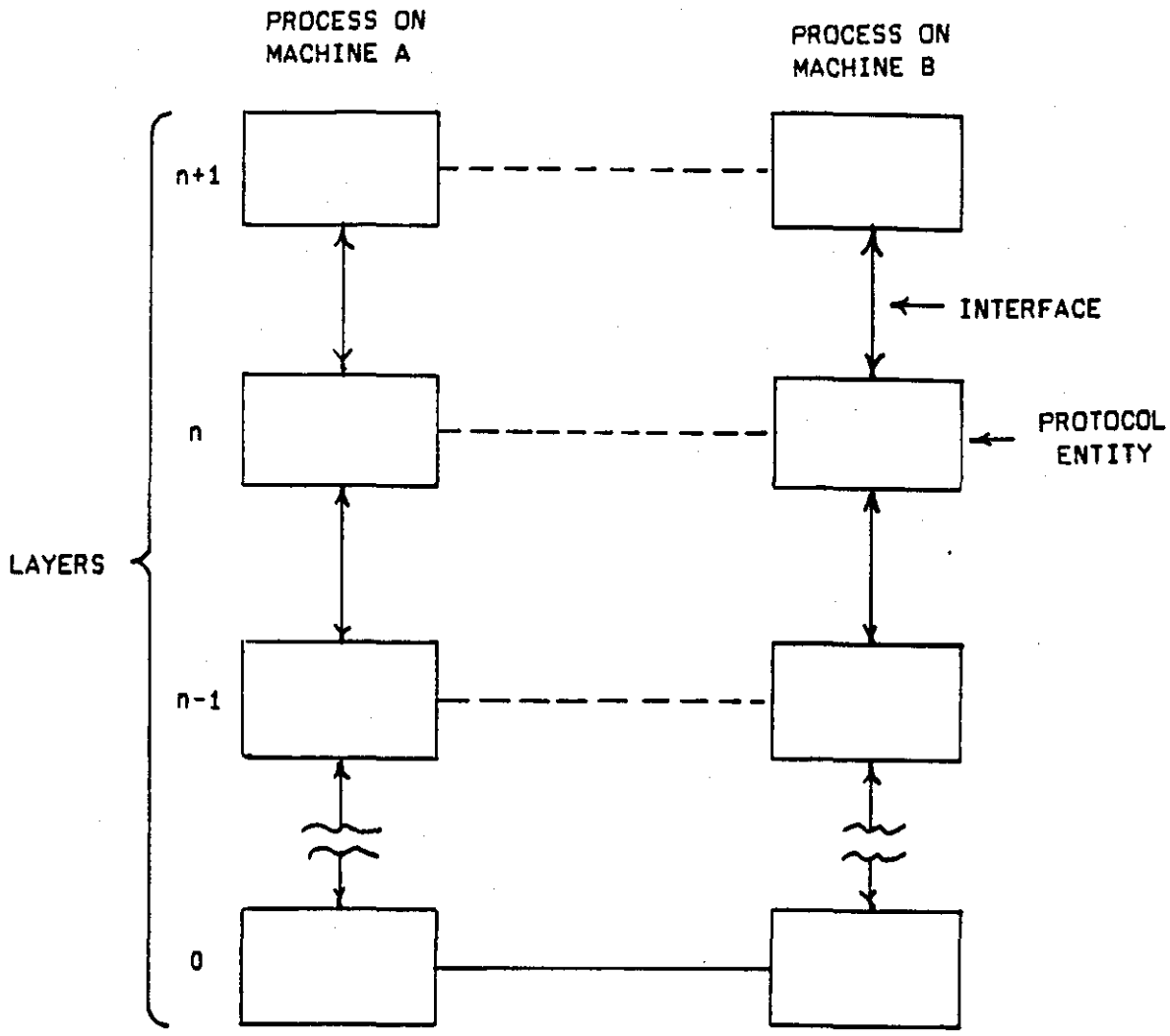


FIGURE 1.2 - LAYERING OF PROTOCOLS

(1) THE PHYSICAL LAYER

This is concerned with transmitting raw bits over a communication channel.

(2) THE DATA LINK LAYER

The task of this layer is to take the transmission facility provided by the physical layer and transform it in such a way that it appears free from transmission errors to the network layer.

(3) THE NETWORK LAYER

This layer controls the operation of the communications subnet. It deals with the routing of messages through the network.

(4) THE TRANSPORT LAYER

This layer accepts data from the session layer, splits it up into smaller units and passes them to the network layer and ensures that pieces arrive correctly at the other end.

(5) THE SESSION LAYER

This layer sets up and manages communication paths between processes and hosts.

(6) THE PRESENTATION LAYER

This layer provides services frequently required on a network such as file transfer, data security and data compression.

(7) THE APPLICATION LAYER

These are the application programs that use the network services.

This information is illustrated in figure 1.3.

1.4.2. IEEE STANDARD 802

With a variety of local area network topologies becoming available a standard was needed to accommodate them. The IEEE Standard 802 defines a family of communication protocols for bus and ring LANs. The basic approach was to split the data link layer of the ISO model into two parts: the network access method, as dictated by the LAN type, and the logical link control independent of the particular network technology used. The logical link

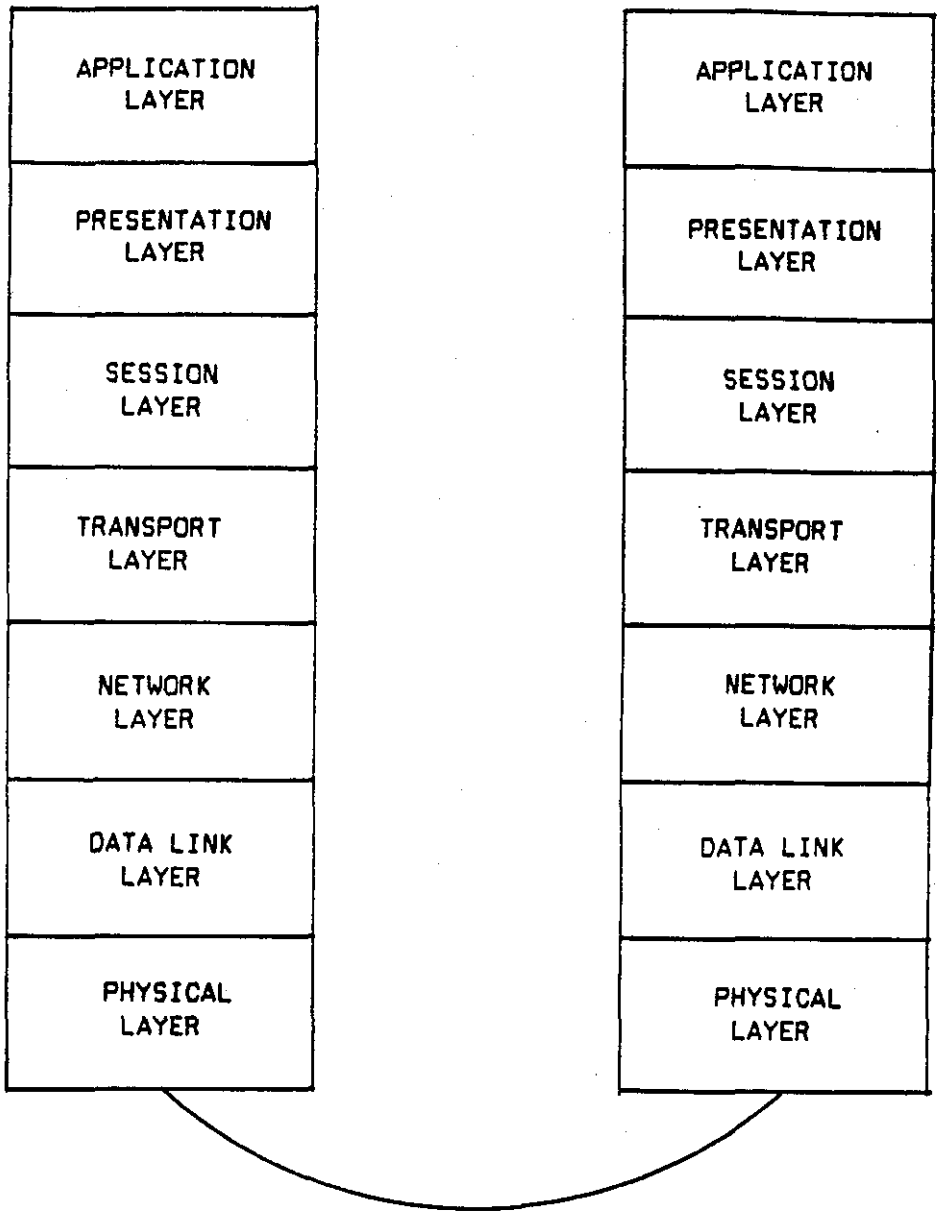


FIGURE 1.3 - THE ISO REFERENCE MODEL

layer provides services similar to those provided by the High-level Data Link Control (HDLC) standard that has been adopted for the data link layer of the ISO model. This approach is illustrated in figure 1.4.

1.4.3. SYSTEMS NETWORK ARCHITECTURE

The Systems Network Architecture (SNA) has been developed by IBM to allow its customers to construct their own networks. What follows is a brief introduction to SNA, a fuller discussion is found in Schultz(1980). SNA can be viewed as a five layer model:

(1) DATA LINK CONTROL LAYER
The takes the raw transmission facility and makes it appear error-free. Thus has the same function as the data link layer in the ISO model.

(2) PATH CONTROL LAYER
This layer manages routing and flow control throughout the network.

(3) TRANSMISSION CONTROL LAYER
This layer creates, manages and deletes end-to-end connections.

(4) DATA FLOW CONTROL LAYER
This layer is primarily concerned with maintaining the correct sequence of data across a connection.

(5) NETWORK SERVICES LAYER
This layer provides the user interface to the network and encompasses the functions of both the session and presentation layers of the ISO model.

This is illustrated in figure 1.5.

1.5. SERVICE SPECIFICATION

Between each pair of adjacent layers in any of these reference models there is an service definition. This defines the primative operations and services the lower layer offers to the upper layer. Each layer uses the service, provided by the layer below, adds some functionality of its own, and thus provides a more con-

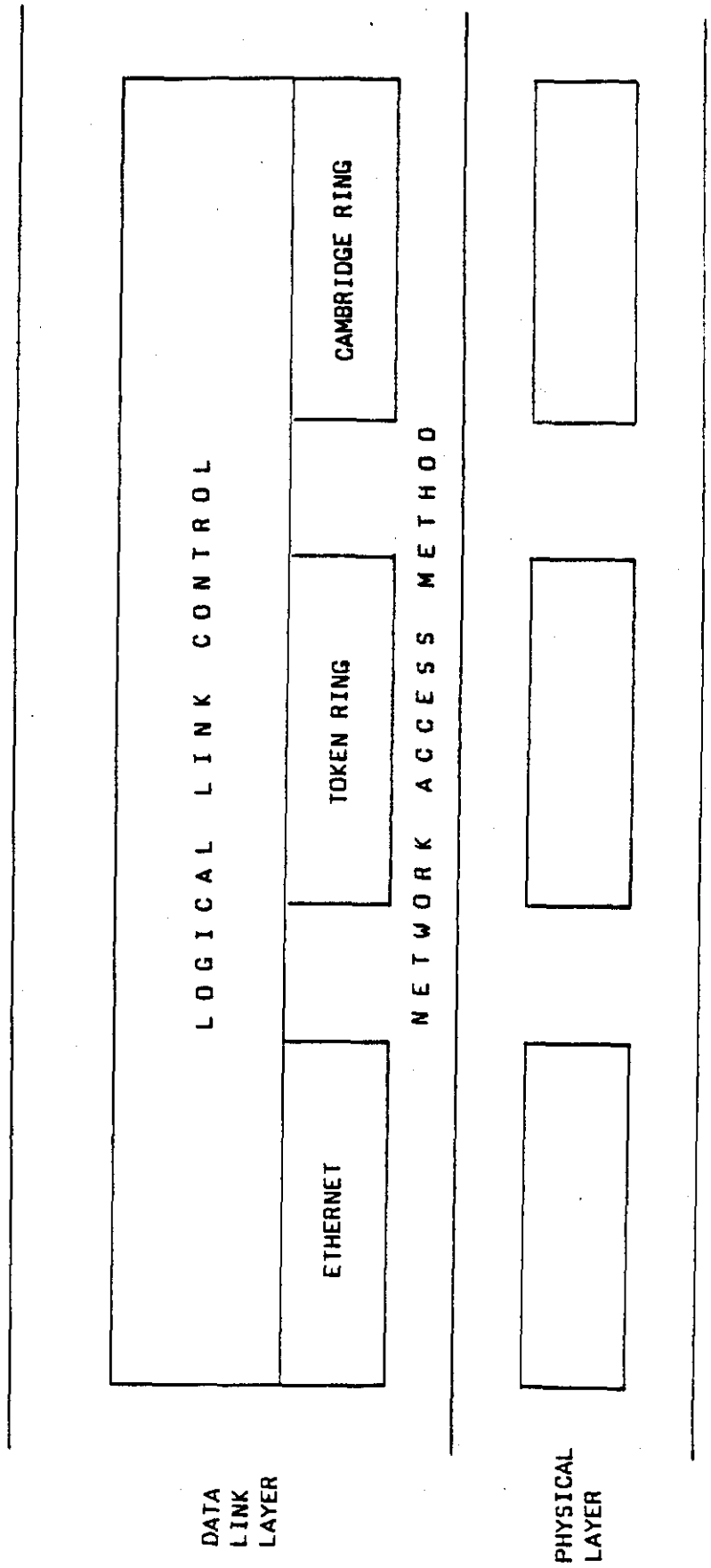


FIGURE 1.4 - THE IEEE STANDARD 802

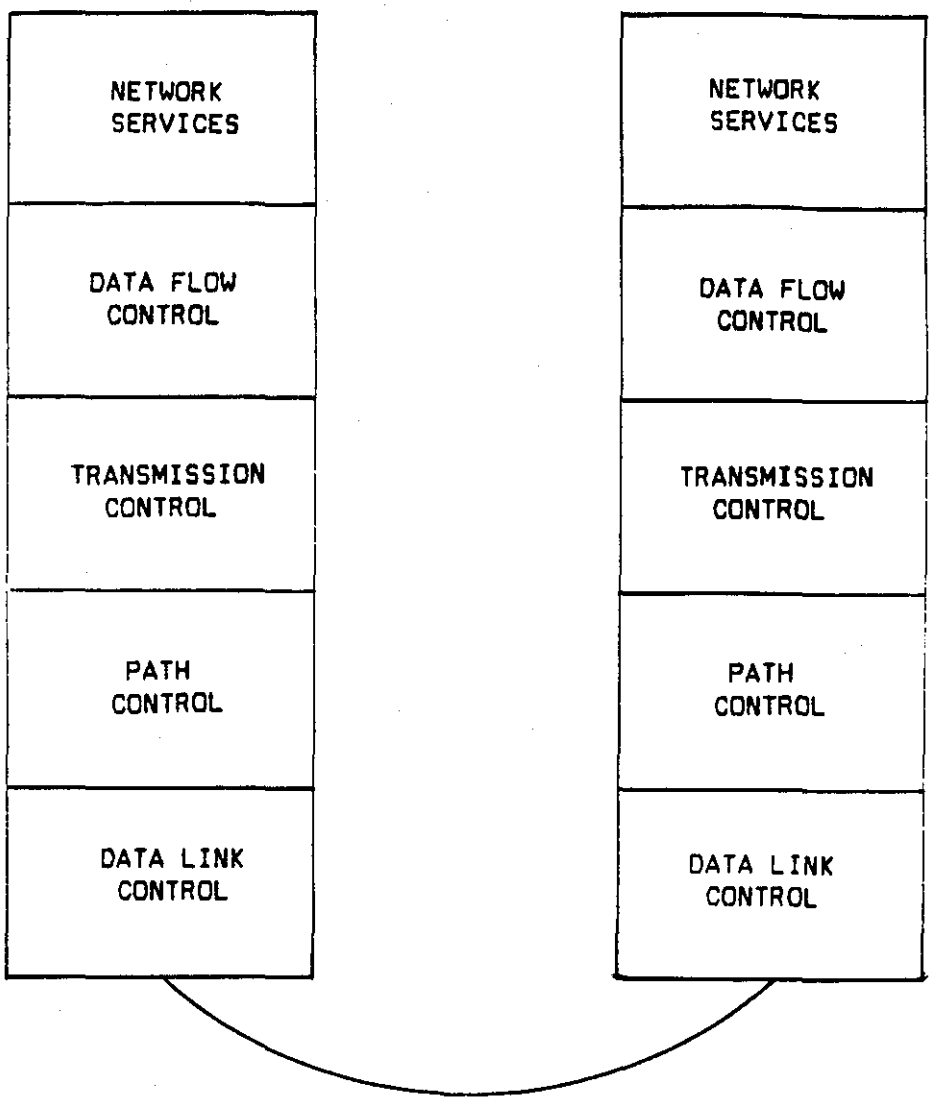


FIGURE 1.5 - THE SNA REFERENCE MODEL

venient interface that can be used to construct a higher layer. The upper most layer provides services for direct use (through appropriate software interfaces) by a "user". The "user" may be a person using network facilities via an operating system command language or a process running under the control of the operating system.

1.6. PROTOCOL SPECIFICATION

The specification of the actions of the protocol entities within a particular layer is called the protocol specification. These actions will be taken in response to external stimuli such as commands from the layer above and messages from the layer below.

Protocol specification is the subject of the next chapter and will be discussed in detail there.

1.7. ERROR AND FLOW CONTROL

Several techniques are widely used to overcome transmission errors and control the flow of packets of information between peer entities.

The reasons why transmission errors occur will be examined first. Following this, there will be an examination of the effect that these errors have on blocks of data being transmitted through a network. Finally, methods for error and flow control will be discussed.

1.7.1. TRANSMISSION ERRORS

Other pieces of electronic equipment, power lines and faulty power supplies can interfere with transmissions. Such interference is often referred to as noise. Noise tends to come in bursts, that is, it effects a string of bits, rather than individual bits in isolation. This characteristic of noise has both advantages and disadvantages when it comes to error control. On the advantage side, since the data is usually sent in blocks of bits, only a few blocks will be effected by the occasional burst error. Suppose the block size is 1000 bits and the error probability is 0.001 per bit. If errors were independent, most blocks would contain an error. However, if errors come in bursts of 100 bits,

only one or two blocks in every hundred would be effected. The disadvantage of burst errors is that they are harder to detect and correct than isolated errors. Studies of protocol efficiency, such as Field(1977), have traditionally considered both independent and burst errors.

1.7.2. ERROR CONTROL

Before we can eliminate errors, we must first detect that they have occurred. Errors can be detected by adding redundancy in the form of checksums. A checksum is an additional field added to the end of a block of data. It is calculated by an agreed formula from the contents of the block. The checksum can be recalculated at the other end of a transmission and if it is incorrect a transmission error has occurred. Checksums may be either error-detecting or error-correcting. Error-detecting codes are usually chosen because error-correction techniques can not cope with total loss of a message and only work in situations where the probability of error is low.

A popular form of error detection is the Cyclic Redundancy Check (CRC). The checksum is calculated by dividing the message, treated as one long bit stream, by a constant divisor. The receiver repeats the division, compares the locally generated remainder with the received CRC and accepts the message if they are identical. The length of the remainder, and hence of the CRC, depends on the divisor that is used. Given a suitable divisor the probability of an undetected error can be made very small.

The receiver can detect whether a packet has been transmitted correctly. In addition, the sender needs to know the result of his transmissions to decide whether to retransmit if an error has occurred. This is usually done by an acknowledgement system. The receiver sends some form of acknowledgement for each packet he correctly receives. He sends a positive acknowledgement to inform the sender that the packet has been successful transmitted. In some systems he can also send a negative acknowledgement if a checksum error has been detected.

It is possible that a noise burst could destroy one or more whole packets in transit. Some agreed form of reference is

required between sender and receiver to identify each message. This is because duplicate and missing messages must be detected by the receiver. Hence each data block is usually proceeded by a header containing a sequence number. The header, data block and checksum together form a data frame.

A popular system of error control is known as positive acknowledgement and retransmission (PAR). In this system the sender sends a data frame and waits for a positive acknowledgement from the receiver. If one is not received within a certain time limit the frame is retransmitted. This system guards against the loss of acknowledgements as well as loss of data frames. Since the sender waits for each transmitted message to be acknowledged before sending another, this type of protocol is also known as a send-and-wait protocol.

1.7.3. FLOW CONTROL

A PAR system also controls the flow of data between sender and receiver. Since the sender waits for each message to be acknowledged before proceeding, a slow receiver cannot be swamped by data from a faster sender. However, if the propagation delay is high very little of the bandwidth is actually used. Hence, the idea of a sliding window protocol was devised.

Under a sliding window protocol the sender is allowed to send a number of data frames up to an agreed maximum. The set of unacknowledged frames is called the send window. As the frames are acknowledged at the bottom of the window, the window "slides" allowing more messages to be sent.

This type of protocol system can be implemented in several different ways. The differences result from varying strategies that can be employed to handle packet loss and corruption. Two strategies will be outlined here.

(1) GO BACK N

Under this strategy, if a frame in the send window remains unacknowledged for longer than the timeout interval, that frame and all other frames sent after it are retransmitted. On the receiver side all frames after a checksum error and

those arriving out of sequence are discarded until the next frame in the sequence is received with no errors. The receiver maintains a receive window only large enough to hold a single packet.

(2) SELECTIVE REJECT

Under this strategy, as soon the receiver detects a missing packet or checksum error it sends a negative acknowledgement to the sender. However, the receiver continues to collect all the good messages following the bad one in a receive window. It waits till the missing message is received and then forwards the contents of the receive window to the layer above. The receiver can maintain a receiver window equal to the size of the send window used by the sender.

A fuller discussion of these issues can be found in Tanenbaum(1981).

1.8. SUMMARY

In this chapter some of the basic terms of computer networks have been defined and some key concepts have been outlined. The various components of a network architecture have presented together with some examples of reference models. Finally, the important topics of error and flow control were discussed.

CHAPTER TWO

PROTOCOL SPECIFICATION

2.1. INTRODUCTION

Computer communication has many parallels with human communication. Both consist of an exchange of messages and both require a language known to both parties. The study of human language is known as linguistics. Those interested in the area of comparative programming languages have used many terms originating in linguistics in their work. Thus the words grammar, syntax and semantics are known to most people working in computer science. Unfortunately, workers in the field of communication protocols have adopted a different set of terms which are not generally understood.

For example, Davies(1979) differentiated between the logical and procedural specification of a protocol. In language there are only two basic mechanisms for conveying information. One way to convey information is through the content of various units, be they words, sentences, flowchart symbols or protocol packets. The other way is by arranging these units according to some set of rules. For example, the sentences "the cat sat on the mat" and "the cat mat on the sat" contain an identical set of words, but the first is constructed according to the rules of grammar while the second is not.

The logical specification is concerned with the first mechanism, the format and meaning of messages in the protocol language. For example, the third and fourth byte of a message may give the source of a message and the last two may be a checksum. The message as a whole may be a block of data or have some other meaning.

The other mechanism is the subject of the procedural specification. This is concerned with the interaction of peer entities which takes the form of an exchange of messages. The rules governing the sequence of the messages exchanged within a given protocol are sometimes known as the rules of procedure.

Both specifications are concerned with syntactic and semantic issues. Brown(1984) describes a useful distinction between literal, functional and pragmatic meanings as they relate to human language:-

The literal meaning of an utterance is its meaning taken in isolation from any context in which it is spoken.

The functional meaning is the meaning intended by the speaker and the purpose behind the utterance.

The pragmatic meaning is the meaning derived by the hearer in a particular context which may result in a certain course of action on his part.

The statement "It is raining" may be a simple statement of about the weather; its literal meaning. In response to the question "shall we go out for a walk?", it may have be a way of politely declining the invitation, and this would be a functional meaning. However, if this statement is spoken to a housewife with washing drying in the garden this statement may have the meaning "My washing is getting wet!" and cause her to go outside to bring it in, even though the speaker may not have intended this to happen. This would be a pragmatic meaning.

In the same way message number 34 followed by a valid check sum is on the surface a simple data block. If the last message received by an entity was numbered 33 the entity may simply pass the text to the layer above. Alternatively, if the last message received was numbered 32 this may imply a message has been lost and result in completely different actions on the part of the receiver.

The literal meaning of a message can be founded by referring to the logical specification of the protocol, while the functional and pragmatic meaning can only be derived by referring to both the procedural specification and the history of the current exchange.

This last statement introduces us to the fact that a protocol entity needs to maintain some information resulting from previous transactions. This information is usually called the state of the entity. The process of changing states is known as a transition. Thus, the most popular forms of protocol specification method are known as state-transition methods.

However, before a discussion of these methods can take place, the basic protocol data units must be defined. This is the task

of the logical specification.

2.2. LOGICAL SPECIFICATION

The specification of messages is relatively straightforward. We need to consider what is the lowest level of data representation we are to consider. If the data units are expressed in terms of binary, it is a bit-oriented protocol. Alternatively, if they are expressed in terms of characters, it is a character-oriented protocol.

The format of a data unit or frame can be presented in the form of a block diagram as in figure 2.1. Such a representation is equivalent to a record structure found in high-level programming languages as in figure 2.2. Alternatively, a grammar notation, such as Backus-Naur form, could be used as in figure 2.3. This later form is especially useful where there are classes of frames with a similar structure, as is found in HDLC.

2.3. PROCEDURAL SPECIFICATION

There are many different approaches to procedural specification in the literature. Harangozo(1977) describes a protocol by specifying the set of all legal exchanges using grammars. This level of abstraction is useful in the design stage. However, it is not very useful for those interested in producing software to implement the protocol. This is because complex processing is required to translate this type of specification into algorithmic form. Hence, most authors have specified protocols by describing protocol entities which conform to the rules of procedure of that protocol. Examples of this approach can be found in Bochmann(1977a & b) and Alfonzetti(1982).

These two approachs are complementary, since both forms of specification will be required at different stages of the development of a protocol. The specification methods in the next section can, in general, be applied in both these approaches.

FIGURE 2.1 - A BLOCK DIAGRAM

```
record
   id   : 0..1;
   seq  : 0..7;
   data : array [1..10] of char;
   checksum : 0..255;
end
```

FIGURE 2.2 - A RECORD STRUCTURE

<frame> ::= <identifier><sequence number><data><check sum> .

<identifier> ::= 1 | 0 .

<sequence number> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 .

<data> ::= <character list> .

<character list> ::= <character list><character> .

where <character> is an ASCII character
and <check sum> is a single byte.

FIGURE 2.3 - GRAMMAR FORM

2.4. A SURVEY OF PROTOCOL SPECIFICATION METHODS

2.4.1. INTRODUCTION

Many surveys describing various specification methods have been published. These include Bochmann(1980), Danthine(1980), LeLann(1978), Merlin(1979), Stenning(1979) & Sunshine(1978,1979). The first protocols were described chiefly in prose with a few diagrams of frame structures. Unfortunately, the ambiguity inherent in natural languages lead to the specifications being interpreted in different ways by different implementors. Thus various types of tables and diagrams were employed to enhance the prose description. A good example of this type of specification is the document "Cambridge Ring 82 - Protocol Specification" (Larmouth,1982). This contains prose description, frame descriptions using BNF notation and time sequence diagrams to show possible message exchange sequences. These methods were a great improvement on straight prose, but there has been a desire to introduce a much greater degree of formality into specifications. Greater formality would enable a protocol specification language to be developed which could be used in protocol design and implementation tools.

There are two main types of formal specification methods identified in Piatkowski(1983).

(1) State-transition methods in which input/output behaviour of a system is defined indirectly by specifying a state variable, possibly with a number of components, and a series of transitions involving input/output.

(2) Sequence expression methods in which input/output behaviour is defined directly without recourse to internal state variables.

These two approaches will be considered in turn. In addition, temporal logic will be discussed. This is an extension to boolean algebra useful in protocol specification.

2.4.2. AN EXAMPLE PROTOCOL

An example protocol is introduced for subsequent discussion. The protocol is at the data link level of the ISO model. As a simplification an error-free transmission medium is assumed. Therefore, the frames have no checksum and timeouts have not been included. The physical layer provides a half-duplex link, which means that frames can be transferred in both directions but not simultaneously. Flowcharts of a suitable protocol are given in figure 2.4 and 2.5.

2.4.3. STATE TRANSITION METHODS

There are two main state transition methods: Finite State Machines and Petri Nets. Firstly, the classical versions of these techniques will be introduced together with applications to the example protocol. Secondly, several extensions to these methods will be discussed which increase the power of these techniques and make automatic implementation possible.

2.4.3.1. FINITE STATE MACHINE

A (deterministic) finite state machine M consists of a set of 5 components (Cooke,1984).

a) Q a non-empty set of states.

b) A a finite alphabet.

c) t a mapping Q x A -> Q of transitions.

d) q_0, a member of Q, the initial state.

e) F, a subset of Q, the set of final states.

In a protocol specification the alphabet, A, is a set of events such as sending a message, receipt of a packet or timeout. The set of states, Q, can be either the state of an individual entity or the composite state of a pair of entities and the underlying medium. The machine, M, can be represented by a diagram. The elements of Q are represented by nodes on a directed graph. Each member of Q is drawn as a small circle enclosing the state name. Elements of F have an additional circle drawn around then. For each element $((q_i,a_j),q_k)$ in t, an arc is drawn from q_i to q_k

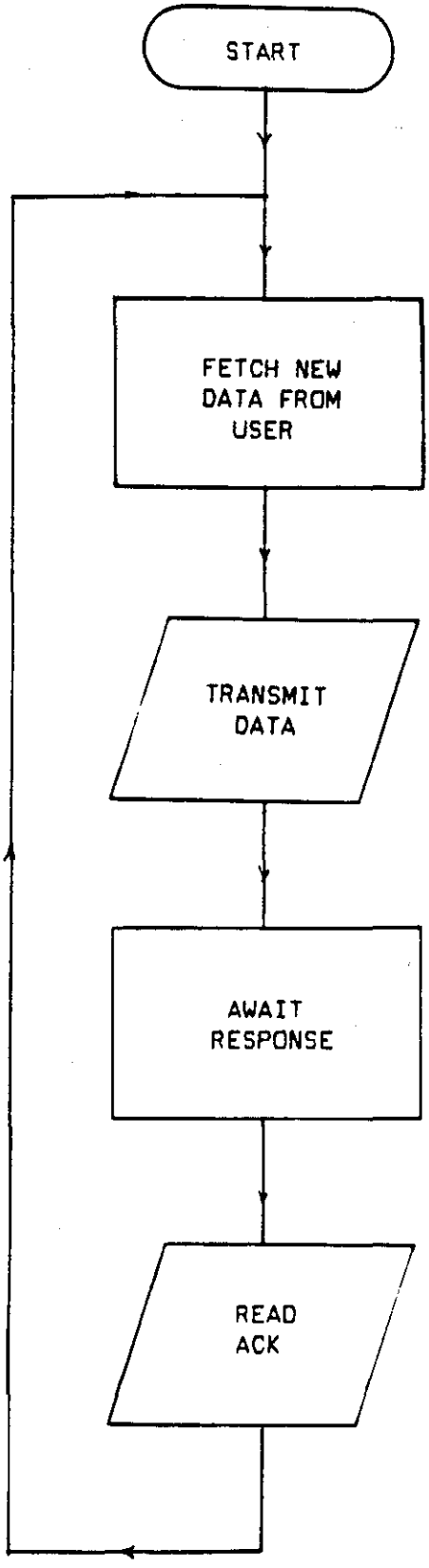


FIGURE 2.4 - AN EXAMPLE PROTOCOL (SENDER)

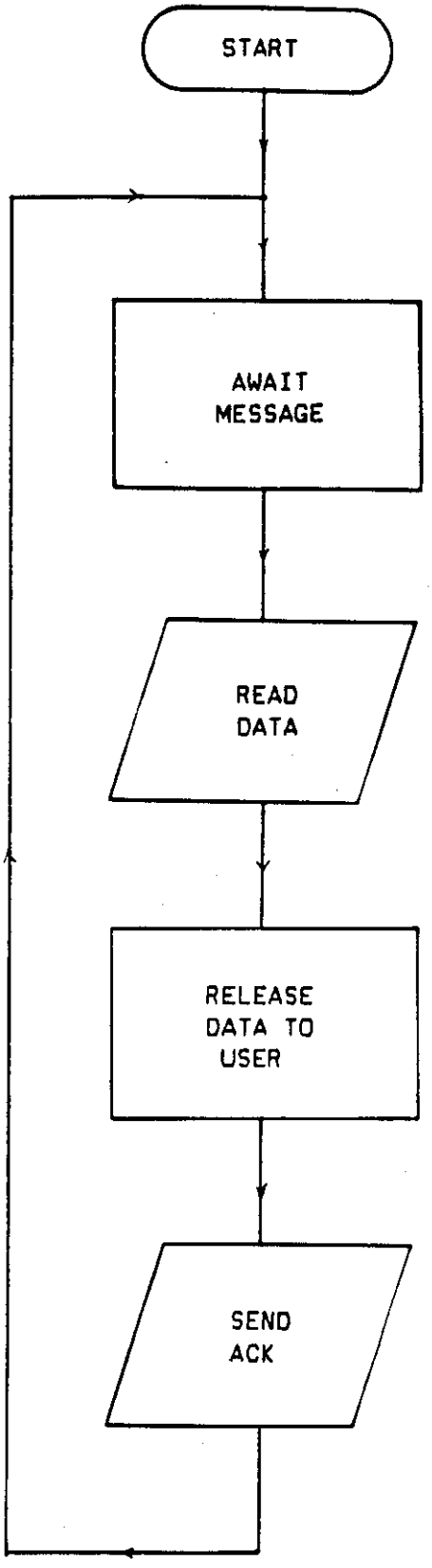


FIGURE 2.5 - AN EXAMPLE PROTOCOL (RECEIVER)

which is labeled a_j. q_0 is identified by an arrow pointing to it.

As an example of this method, state transition diagrams for the example protocol are given in figures 2.6. In this case F is the empty set, ie there is no final machine. In addition, the characteristics of the communication medium must be defined. This can be done using a third finite state machine. This machine is illustrated in figure 2.7.

The overall system is now modeled by three finite state machines. Since a transition on one machine may cause a transition on another, the transitions on these machines are interdependent. For example, the transition SEND DATA on the sender is related to the transition CARRY DATA on the communications medium, which is itself is related to the transition READ DATA on the receiver. Any two transitions which are related in this way are said to be directly-coupled. Furthermore, any two machines with directly-coupled transitions can also be said to be directly-coupled. For example, the sender entity is directly-coupled with the communications medium, and so is the receiver. However, the sender and receiver are not directly-coupled since there is no direct coupling of the transitions of these machines. Transitions which are caused by a transition on another machine are said to be dependent transitions. Transitions which are not dependent are said to be spontaneous.

If we consider all transitions to be atomic, it is possible to combine these three machines into a single composite machine using a fairly simple procedure. Firstly, the first state of the composite machine is defined to be a tuple made up from the initial states of the three machines. Hence it is written (1,1,1). The next step is to examine each machine for a spontaneous transition starting from state 1. The only possible transition is the FETCH DATA transition on the sender machine. Hence the next composite state is (2,1,1). Each machine must now again be inspected for spontaneous transitions, and also for transitions directly-coupled with with FETCH DATA. The only possible transition is the spontaneous transition SEND DATA on the sender machine. Hence the next composite state is (3,1,1). At this stage there are no spontaneous transitions. However, the transition CARRY DATA on the

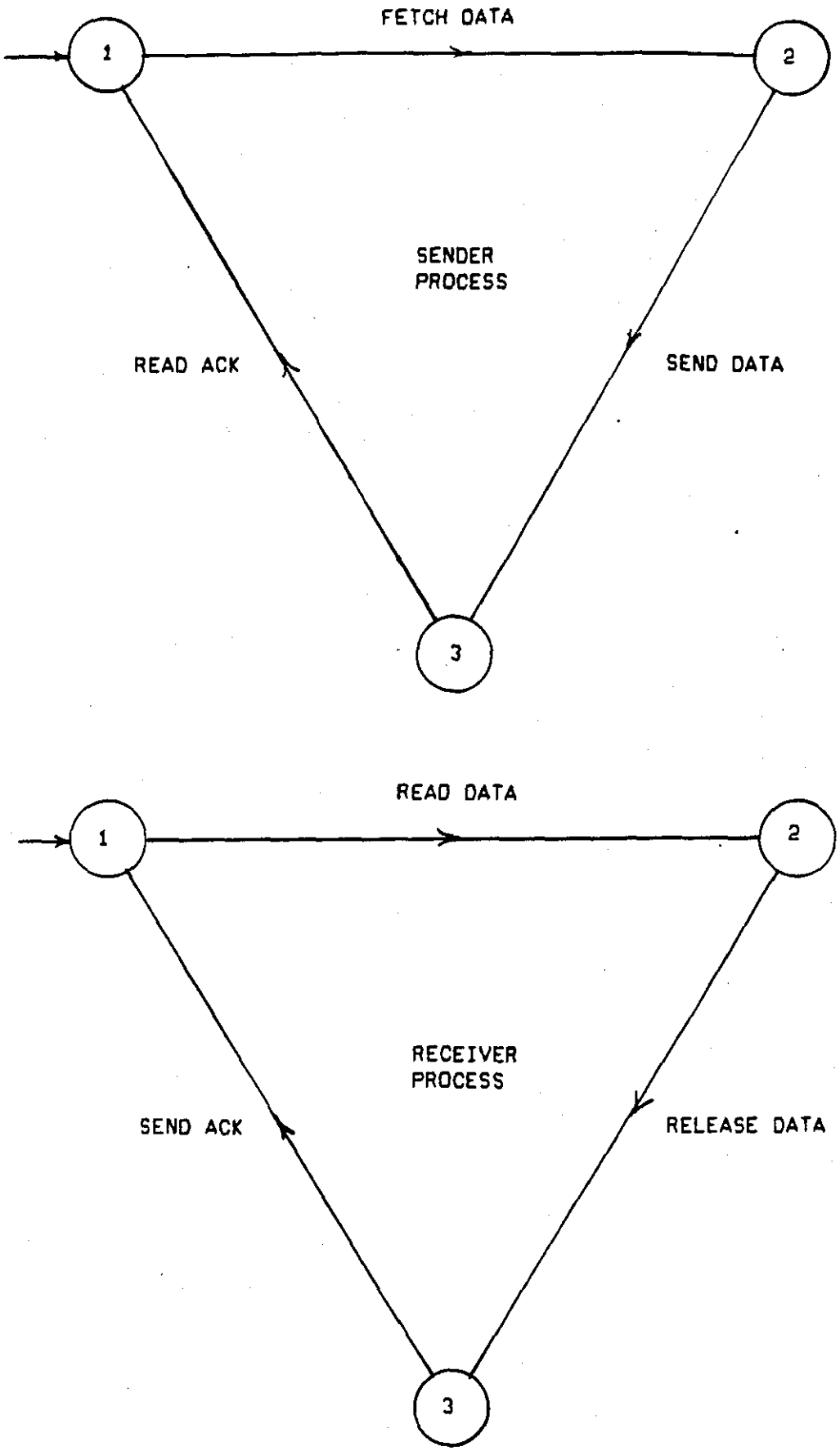


FIGURE 2.6 - FINITE STATE MACHINE FOR PROTOCOL ENTITIES

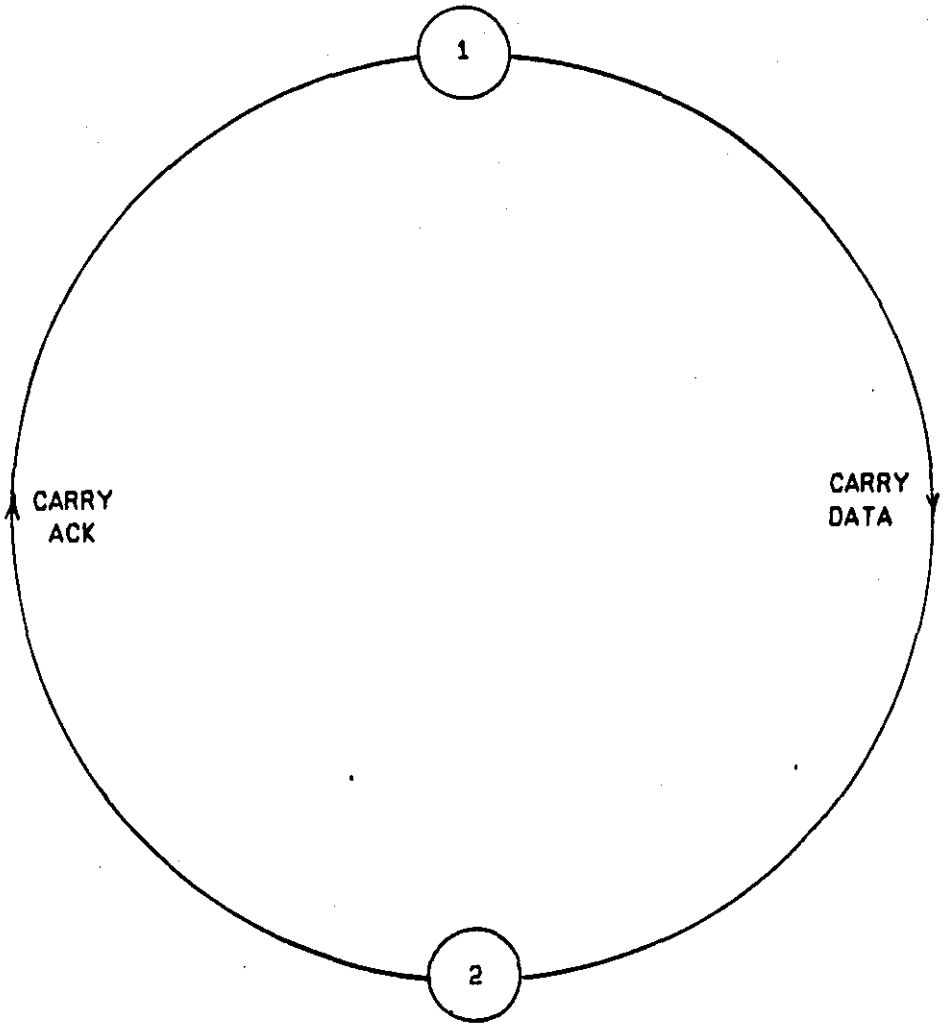


FIGURE 2.7 - FINITE STATE MACHINE FOR COMMUNICATION MEDIUM

communication medium is directly-coupled with the SEND DATA transition on the sender. Thus the next state is (3,2,1). This procedure continues until the state is once again (1,1,1) and all possible paths back to that state have been explored. Figure 2.8 shows the complete composite machine for the example protocol.

Composite machines like this are useful for validating protocols. For example, a node with no successors indicates a possible deadlock.

2.4.3.2. PETRI NETS

An alternative method of modeling protocols is the use of Petri Nets (Diaz,1982). A basic Petri Net C consists of a set of four elements (Peterson,1977).

a) P, the set of places.

b) T, the set of transitions.

c) I, the input mapping T -> 2^P, ie., the set of input places for each transition.

d) O, the output mapping T -> 2^P, ie., the set of output places for each transition.

The Petri Net C can be represented by a diagram. Places and transitions are represented by nodes on a graph. A place is denoted by a circle and a transition by a short line. I and O are represented by directed edges. Whereas in a finite state machine the nodes represent states and the edges represent possible transitions, in a Petri Net possible transitions are represented by transition bars, and state information is represented by the presence of tokens at places. The state of a net is given by the token distribution known as the marking. Formally, a marking is a mapping of the set of places into the set of natural numbers, diagrammatically it is shown using dots to represent tokens. These dots are placed in the circles denoting the places. A transition can fire (occur) when each of its input places holds at least one token. When the transition fires it removes a single token from its input places and deposits a single token at each of its output places.

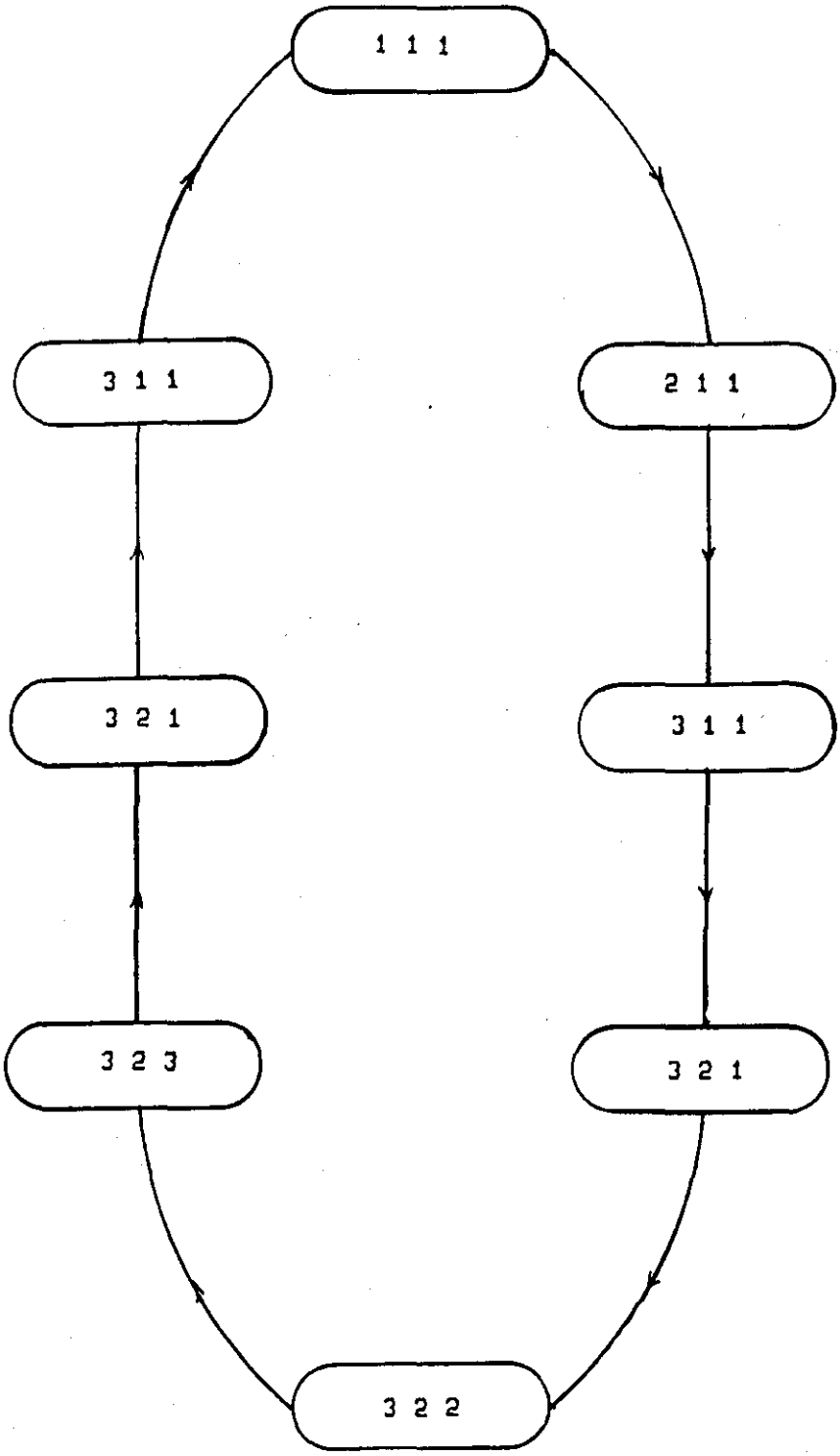


FIGURE 2.8 - COMPOSITE MACHINE FOR EXAMPLE PROTOCOL

Consider the simple Petri Net in figure 2.9. The marking indicates that the initial state has a single token at place S. The presence of this token indicates that transition 1 can fire. When the transition fires it removes the token from place S and deposits one token in place A and one token at place B. Transition 2 can now fire since there is a token at both place A and place B. These tokens are now removed from these places and a single token is deposited in place X. Thus the final marking is a single token at place X and no tokens in any other places.

A Petri Net can conveniently be used to represent a protocol. Certain places are used to represent discrete states of the individual entities. The presence of a token in one of these places indicates that a particular entity is in a certain state. Other places represent particular frame types and the presence of a token at such a place indicates that a frame is in transit in a particular direction. If several of that type are in transit simultaneously then several tokens will be present at that particular place.

A Petri Net for the example protocol is given in figure 2.10. The places at the left of the Net represent the states of the sender entity, the places at the right represent the states of the receiver entity, and the places in the centre represent frames in transit. A single diagram has been used to model the structure of the system. Starting at the models initial marking we can construct a finite state machine to model the behaviour of the system. Such a machine is called a token machine. Its structure is the same as the composite machine in figure 2.8.

2.4.3.3. SYNTHESIS

The discussion of state transition methods began with a definition of deterministic finite state machine. This definition was used together with some other concepts to show how a collection of linked finite state machines can model a protocol layer. A construction was then outlined to combine these machines into a single machine for protocol analysis. Following this the classical Petri Net was described.

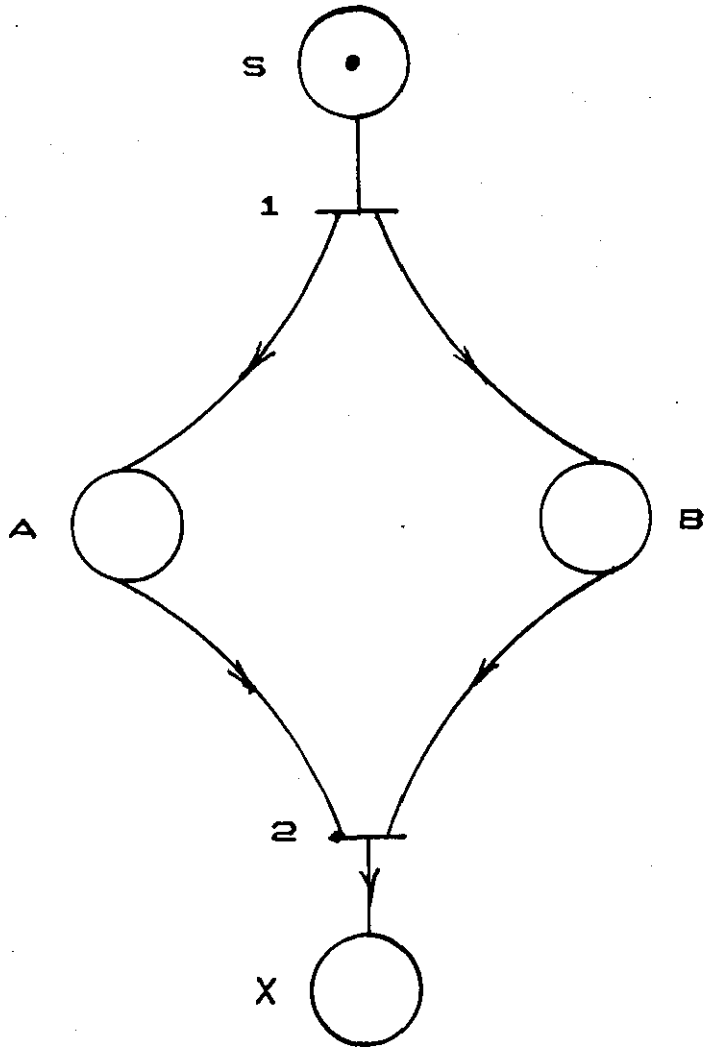


FIGURE 2.9 - A SIMPLE PETRI NET

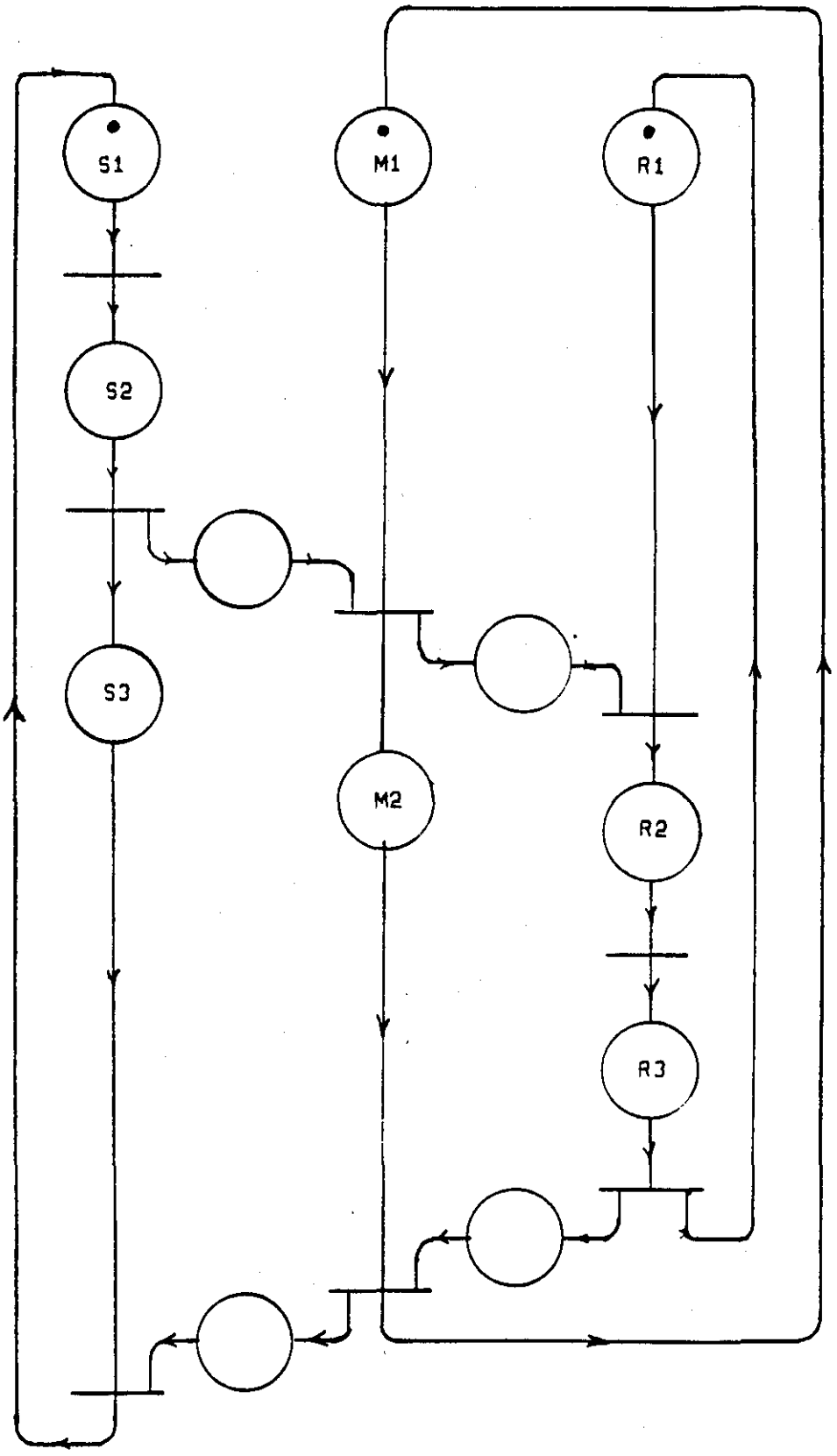


FIGURE 2.10 - PETRI NET FOR THE EXAMPLE PROGRAM

It will be noted that the classical Petri Net contains no equivalent to the alphabet in the finite state machine. This is because the firing of transitions is only dependent on the current marking. However, each transition can be labelled with the name of the event, or events, it represents. A transition may, for example, represent the entity sending a particular frame type to a peer. In this case two parts of the system, an entity and the underlying communications medium are involved. Such an interaction is equivalent to a directly coupled transition of a system of directly-coupled finite state machines. A Petri Net can be decomposed into separate Petri Nets with coupled transitions. This is achieved by allowing an optional condition or predicate to be added to each transition. In this case a transition that can fire as a result of the current marking will only fire when this predicate is met. Such a predicate may represent the reception of a particular frame type or the availability of data. Hence it can be seen that finite state machines are in fact a subclass of Petri Nets suitable for modeling sequential processes.

The choice of a suitable representation of protocols will depend on the particular application envisaged. This thesis is concerned with automated protocol implementation so the representation chosen has to be suitable for this work. The representations discussed so far are not suitable for input to a computer. This is partly due to their graphical nature, but is also due to incompleteness. Hence, various extensions need to be considered.

2.4.3.4. EXTENDED STATE TRANSITION METHODS

The basic strategy adopted in the literature has been to expand the definition of a transition to include a programming language style description. Keller(1976) proposed a model of this form for representing parallel programs. His model consists of a Petri Net complemented with a set of variables X. Each transition t has associated with it an enabling predicate P_t, depending on some variables in X, and an action A_t, assigning new values to some variables in X. The state of the modelled system is determined by the number of tokens that reside in different places and the value of variables. A certain transition is said to be enabled, that is it can fire, when all its input places have at

least one token and its enabling predicate P_t is true. When a transition fires the corresponding action A_t is executed and tokens are redistributed according to the rules of Petri nets.

This model has following characteristics.

(1) The control structure is represented by the interconnection of places and transitions, and some variables of the set X.

(2) The semantic structure is represented by the variables, predicates and actions associated with the transitions.

Bochmann(1977b) adapted this approach and used it in protocol specification. A protocol layer can be modeled as a system of extended finite state machines. An extended finite state machine is a finite state machine complemented by variables, predicates and actions according to Keller's approach. Each protocol entity will contain:

a) Definitions of variables

b) A finite state machine

c) A collection of associated predicates and actions.

Ayache(1982) further refined this approach by introducing an additional type of predicate called the reception predicate RP_t. If a reception predicate is associated with a transition it can only fire if the message type or types specified by the predicate are received by this entity. Therefore, an extended finite state machine can be written as a list of transitions in the form:

(pre-state),(reception predicate),(predicate)
-> (action),(post-state)

where the pre- and post- state are the names of states in the finite state machine. It can be noted that this is similar to the standard form of an operation presented in Jones(1980).

(pre-condition) -> (action),(post-condition)

Extended state transition methods represent a good basis for the design of a protocol specification language. There are, however, other approaches which must be considered before these methods are

discussed in greater detail.

2.4.4. SEQUENCE EXPRESSION METHODS

An alternative approach to the problem of describing the behaviour of system has been devised by Milner(1980). He calls this system a Calculus of Communicating Systems (CCS).

2.4.4.1. CALCULUS OF COMMUNICATING SYSTEMS

The following discussion is based on Milner's work, although some of the terminology has been changed to relate more closely with that used in the rest of this Thesis. A system can be decomposed into a number of parts or entities. Activities within entities are called actions and actions involving two entities are called events.

CCS allows us to model the execution of an entity or process by describing the sequence of events. Since we shall again consider events to be atomic, parallelism can be modeled by an arbitrary interleaving of events. A process can be modeled by a tree-like object with labelled edges. The nodes of a tree represent the process state while the edges correspond to events, as in fig 2.11.

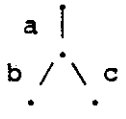


Figure 2.11 - A CCS tree.

These trees do not model processes perfectly since two different trees can describe the same behaviour as in figure 2.12.

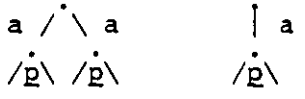


Figure 2.12 - Equivalent trees.

We can describe a number of operations on processes represented in this way.

(1) Sequence (;)

Figure 2.13 - Sequencing.

(2) Choice ([]) (or the alternative composition)

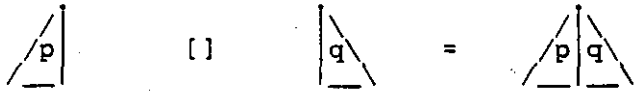
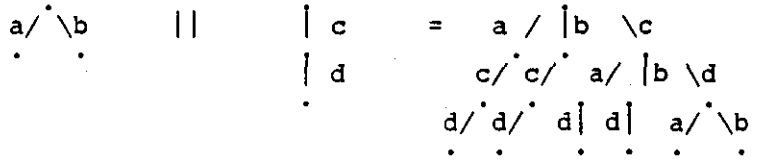


Figure 2.14 - Choice.

(3) Concurrency (||)

a) processes which do not interact



b) processes that may interact

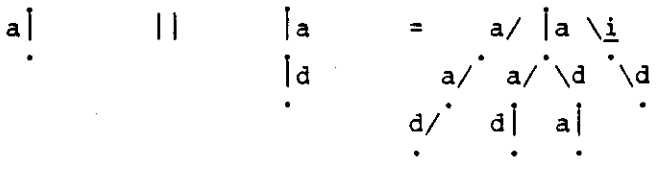


Figure 2.15 - concurrent compositions.

When two processes that may interact are the subject of a concurrent composition there is no constraint forcing them to interact with each other. This is because they may alternatively interact with other processes outside the composition. The label i in the last diagram of figure 2.14 indicates the case where they do in fact interact. This interaction is internal to the composite process and is not "externally visible" and could therefore be deleted from the tree. If we wish to constrain concurrent processes so they are forced to interact we need to use the hiding operator.

(4) Hiding (\)

To exclude other processes from participation in a given set of events these events must be hidden.

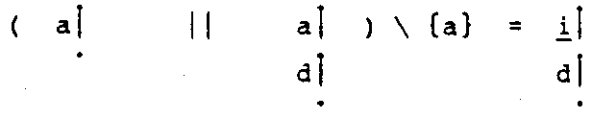


Figure 2.16 - Hiding.

To illustrate the power of this method we shall again refer to the example protocol. The three processes that were earlier represented by a finite state machine interact in the following ways.

a) Sender fetches data from the user.

b) This data is passed to the communication medium.

c) The communication medium passes the data to the receiver.

d) The receiver passes the data to the user.

e) The receiver passes an acknowledgement to the communication medium.

f) The communication medium passes this acknowledgement to the sender.

The three process can be described thus:

S = a;b;f;S
M = b;c;e;f;M
R = c;d;e;R

where S is the sender process, M is the communications medium, and R is the receiver process.

Note that since these processes are non-terminating these expressions are recursive. The expression for deriving the external behaviour of these process is:

C = (S || M || R) \ {b,c,e,f}
C = a;i;i;d;i;i;C = a;d;C .

One of the main advantages of the approach is that the various composition operators facilitate a modular approach, in which processes can be described as compositions of subprocesses.

2.4.5. TEMPORAL LOGIC

Another approach to protocol specification is the use of temporal logic as described in Hailpern(1983) and Schwart(1982). This is basically an extension to the system of boolean algebra. The time dimension is added into the system by means of three additional operators: $\Box$, $\Diamond$ and until. The unary operator $\Box$ (henceforth) on a predicate implies that if the predicate is true in the current state it will remain true for all future states. The unary operator $\Diamond$ (eventually) implies that a predicate is true in the current state or will be true in some future state. Given any two predicates A and B, A until B implies that A must be true until the first state in which B is true. For example (a=1) until (b=2) implies that the value of a will be 1 at least until the b becomes 2. It can be noted that

$$\Diamond P \equiv \neg\Box\neg P$$

and that, strictly speaking, until is the only operator needed since

$$\Box P \equiv P \text{ until false}.$$

Many properties of systems can be stated using these operators. If I is invariant throughout a systems execution, that is, it is always true, this is written $\Box I$. To state that P always causes Q to subsequently occur one writes $\Box(P \supset \Diamond Q)$. If P is satisfied infinitely often this can be expressed as $\Box\Diamond P$. This says that for every point in the computation there is a future point at which P is true.

Temporal logic can be used in a variety of ways depending on the underlying model chosen. As has been previously stated information can be encoded in content of units and also by the sequence in which these units occur. The particular unit involved here are states or events. The differences between the various temporal logic approachs result from the way information is distributed between these two encoding mechanisms. Schwart(1982) discusses

three categories of temporal logic specification.

(1) Bound-State specifications consist of temporal logic assertions based on state representations which have a finite set of possible values.

(2) Unbound-State specifications, as proposed in Hailpern(1983), are based on state representations which have an infinite set of possible values. These values reflect the complete history of the process up to any given point in time.

(3) Event-sequence specifications contain no state component and are expressed on the externally visible behaviour of the entities.

Due to its flexibility, temporal logic is a powerful tool when it is used in conjunction with a more operational approach. Lamport(1983) describes an integrated approach which combines state transition methods and temporal logic assertions.

2.4.6. SUMMARY

The various methods which have been presented here can be assessed on a number of criteria. A specification should, as far as is possible, be implementation independent. This means that the various methods should not be constrained to a less than optimal solution implementation because of the structure of the specification. On the other hand, a specification method that leads the implementor towards an optimal solution may be of considerable benefit. A specification method that supports modularity is to be preferred.

Some types of specification, such as Petri Nets, have well-known analytical properties and can be used for modeling and simulation of a complete protocol system. Others, such as finite state machines, have well-known implementation strategies, but require special composition techniques before analysis can begin.

No particular method seems to have a clear overall advantage over all the other methods. Therefore, it is not surprising that the literature contains details of various formal languages based on many of the methods described. Two of these languages are of

particular note. They are the ESTELLE language produced as part of the work on ISO OSI standards, and the Format And Protocol Language (FAPL) developed by IBM.

2.5. PROTOCOL SPECIFICATION LANGUAGES

2.5.1. ESTELLE

The ESTELLE protocol specification was produced as part of the work of the ISO TC97/SC16/WG1 ad hoc group on formal description techniques. This group was established in October 1978 to devise formal description techniques for Open Systems Interconnection protocols. Three subgroups were formed in February 1981. They are called A,B and C and have the following briefs.

A) Definition of architectural concepts.

B) Finite state machine techniques.

C) Sequence expression techniques.

The chairmen of these groups are Gregor v. Bochmann, Richard L. Tenney and Chris Visser respectively. The work of subgroup B produced ESTELLE as reported in Tenney(1983). ESTELLE is based upon extended finite state machines. In an ESTELLE specification a variable called state must be declared which models the state of the transaction as is perceived by an entity. The finite state machine is represented by a list of conditions in the form mentioned earlier, namely:

```
(pre-state),(reception predicate),(predicate)
        -> (action),(post-state)
```

Each part of the transaction is introduced by a keyword as indicated in the table below.

pre-state	- "from"
post-state	- "to"
reception predicate	- "when"
predicate	- "provided"
action	- "begin", terminated by "end".

In addition, an optional "priority" may be assigned to a transition. If two transitions are enabled, the one with the highest

priority will be used. The actions are expressed in the Pascal programming language.

Further details and an example can be found in Tenney(1983). A similar language based on Petri Nets is suggested in Ayache(1982).

2.5.2. IBM'S FAPL

FAPL is described in Schultz(1980), Pozefsky(1982) and Nash(1983). It is basically an extended version of PL/1, incorporating finite state machines and more powerful data types. Extended finite state machines are presented in a tabular form. Columns are headed with state name and rows are labeled with a series of input conditions. At the intersection of the row whose conditions are all met and the column labeled with the name of the current state there is an indication of the next state. The hypen code (-) indicates nothing is to be done, an integer is a new state, a greater-than symbol (>) indicates an error and a divide sign (/) indicates an impossible sequence of events. There is also an optional action code which is an identifier in parenthesis. An example of a FAPL finite state machine is given in figure 2.17.

Two additions to the data types of PL/1 are supported. These are the entity and the list. A list is a linked structure constructed from entities. Various list processing facilities are provided for manipulating these types. A fuller description of FAPL and examples of its use can be found in the literature.

Despite a rather complex format FAPL has been used successfully for the validating and implementing SNA products. However, its general acceptance by the computing community seems doubtful, since it lacks the elegance of ESTELLE.

2.5.3. LOTOS

The LOTOS protocol description technique was devised as a result of the activities of subgroup C of the ISO working group developing formal description techniques. It is based on Calculus of Communicating Systems as devised by Milner and described in section 2.4.4.1. It also incorportes the abstract data types

STATE NAMES-------> STATE NUMBERS-----> INPUTS	RESET 1	AWAITING 2
S, RQ, FIRST_IN_WINDOW S, RQ, ~FIRST_IN_WINDOW	2(PACRQ) -(NOPAC)	\ -(NOPAC)
R, RSP, PAC	>(PACERR)	1(PACRSP)

OUTPUT CODE	FUNCTION
PACRQ	PI = ~PAC;
NOPAC	PI = ~PAC;
PACERR	CALL LOG ('UNEXPECTED PACING RSP')
PACRSP	PACING_CNT = PACING_CNT+WINDOW_SIZE

FIGURE 2.17 - AN EXAMPLE FAPL FINITE STATE MACHINE

language ACT ONE as described in Ehrig(1983).

The basic constructs of LOTOS allow modelling of sequencing, concurrency and non-determinism in an entirely unambiguous way and can model both synchronous and asynchronous communication. LOTOS may be used to describe the allowed behaviours of a system either with or without describing the particular mechanisms which achieve these behaviours.

Modularity is an important characteristic of LOTOS. A system as a whole is a single process that consists of several interacting processes. These characteristics are, of course, derived from CCS. LOTOS is described more fully in ISO/DP8807(1985).

2.6. SUMMARY

Protocol specification is an area of considerable debate. The main dispute is between protagonists of the traditional state approach and the alternative sequence expression approach. Other methods such as temporal logic have properties which are useful in protocol analysis. The first two formal languages presented were both based on finite state machines. LOTOS, a language based on the sequence expression approach was also described.

As development tools different languages may used appropriately at different stages in the development stage. For example LOTOS is appropriate at the early stages while Estelle and FAPL are appropriate at later stages.

CHAPTER THREE

AN ALTERNATIVE APPROACH TO PROTOCOL SPECIFICATION

3.1. INTRODUCTION

Several protocol specification techniques from the literature have been outlined, and the reason for increasing formality in this area have been discussed.

The techniques described so far concentrate on the procedural specification, treating the logical specification as a separate issue. However, the logical and procedural aspects of a protocol specification are interdependent and a protocol specification should disclose this relationship. Clarity can also be increased by eliminating implementation details, such as buffer management and frame assembly which can be deduced from more fundamental aspects of the protocol.

In order to explore these aspects of protocol specification, it was proposed that a new protocol specification language should be devised. The logical specification could be brought into the main specification and made the central pillar around which the specification is written. A study of various protocols revealed that the packet (or frame) structure of many protocols are hierarchical in nature. In such protocols frames are grouped into classes. In HDLC, for example, there are control, information and supervisory classes. Thus a two-tier system seemed desirable for frame structures.

It was also observed that the simple protocols such as send and wait protocols are a special class of sliding window protocols. In the case of send and wait protocols the size of the send window is one. Thus most protocols can be modelled as sliding window protocols.

The need to maintain state information was discussed in the introduction to the previous chapter. A state is a collection of variables which describe the current state of the transaction between the peer entities as it is understood by a particular entity. Jones(1980) uses a state concept based around a set of variables which could be employed in a specification language.

The language devised with these concepts is called PSL/1. (NB: PSL stands for Protocol Specification Language.)

3.2. PSL/1

This language combines a particular procedural approach with special data types. The packet or frame structure is central to the specification of each protocol. PSL/1 employs a two-tier system of frame structure declarations. Overall class formats are defined and fields within these formats can be redefined within frame declarations.

The state information consists of a set of variables. Two data types can be used for these variables. They are fields and integers. A field is a fixed length bit string which can only be incremented according to modulo arithmetic. The modulo of this arithmetic can be derived from the field length by the formula

$$m = 2^{l}.$$

For example, a three bit field is restricted to modulo 8 arithmetic. Fields are generally used for frame sequence numbers.

An integer is of the type found in most high-level languages. The range of values it can take is dictated by the particular machine on which the protocol is being implemented. They are, however, chiefly used as boolean variables or flags.

In PSL/1 all specifications are expressed as sliding window protocols with both a send and a receive window. The size of these windows is specified in the parameters section of the specification. Also specified in this section are two time intervals. One is the frame timeout interval, that is, the maximum period that a sender will wait for an acknowledgement before retransmitting a frame. Frame timeouts are initiated and handled by the underlying protocol system. The other time interval is for the user-initiated timer. This is under the control of the user and an appropriate timeout action can be specified.

The interface between the protocol layer being specified and the layer above is a pair of bit streams, one for input and one for output. The interface with the layer below is expressed in terms of frames. The stream of bits from above is assembled into a data frame according to the frame structure specification and is then placed in the send window.

The send window is a variable length queue of elements containing a frame copy, a frame identification and a timer-count. Since the structure is a queue, elements can only be added to the back and removed from the front on a first-in first-out basis. The length of the queue will depend on the availability of data from the layer above, but it will be restrained to the maximum size specified in the parameters section. Immediately after it is placed in the queue a frame is sent to the layer below. The timer count in the queue element for the frame is periodically decremented until the frame element is removed from the queue or it reaches zero. Should the counter go to zero the frame copy is sent to the layer below for retransmission.

When a frame is received it is placed in the receive window. The receive window is also a queue, but unlike the send window it is of constant length. Each element contains a buffer for the received frame, a field giving the frame identification of the frame to be placed in this element and an accepted flag which is set when the frame has arrived. The data portions of received frames are passed to the layer above in the correct sequence. The formats for both the send and receive windows are illustrated in figure 3.1.

3.3. EXAMPLES

This generalised model of the operation of a protocol is quite flexible and can be tailored to many different types of protocol. An example of a simple alternating bit protocol is given in figure 3.2. Following the example set in Blumer(1980), the specification is for an entity that will fulfill the role of both receiver and sender. This may result in a certain amount of redundancy in a particular implementation, but this must be balanced against the duplication involved in producing two separate specifications.

The specification begins with a title identifying the specification. This is followed by the parameters section. In this case, both the send and receive window sizes are set to one and the timer interval has been specified at ten. In the next section of the specification the two state variables are defined. There

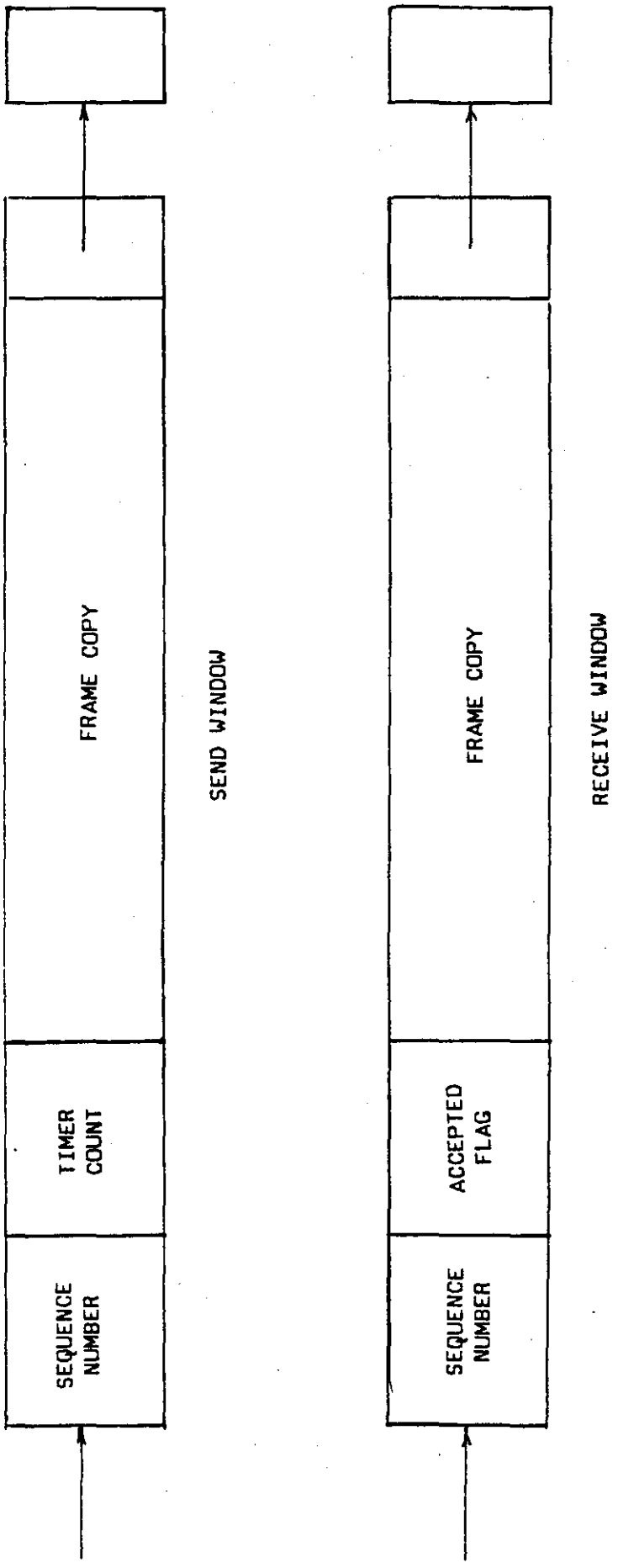


FIGURE 3.1 - SLIDING WINDOW STRUCTURES

```
protocol alt_bit /* alternating-bit protocol */

parameters {
        send_window:=1;          /* size of send window */
        receive_window:=1;       /* size of receive window */
        retran_interval:=..;     /* retransmission interval */
}

state {
        A_seq_num:="0";
        B_seq_num:="1";
}

class control direct {
        format{
                "0";
                seq_num[1];
        }
        frame ack{
                action(receive){
                        if check_sum_error
                        then
                                retran(A_seq_num);
                        else
                                if seq_num=A_seq_num
                                then
                                        cancel(A_seq_num);
                                        inc(A_seq_num);
                                else
                                        retran(A_seq_num);
                                fi;
                        fi;
                }
                action(send){
                        seq_num:=B_seq_num;
                }
        }
}

class info windowed{
        format{
                "1";
                seq_num[1] frame_id;
                inf[10];
        }
        frame dat{
                action(receive){
                        if check_sum_error
                        then
                                /* do nothing */
                        else
                                if seq_num<>B_seq_num
                                then
                                        accept;
                                        inc(B_seq_num);
                                fi;
                        fi;
                        send(ack);
                }
                action(send){
                        seq_num:=A_seq_num;
                        inf:=data;
                }
        }
}

ontimeout{
        single_retran;
}
```

FIGURE 3.2 - AN ALTERNATING BIT PROTOCOL IN PSL/1

is a sequence number for the sending role and another for the receiving role.

This specification describes two classes of frame, the control class and the info class. The control class is specified as being direct. This implies that all frames in that class do not pass through the window mechanism. They are assembled away from the send window and a copy is not kept for retransmission. The info class is windowed, that is it is transmitted via the send and receive window as has previously been described.

Each class is identified by the leading bit, zero indicates a control frame and one an info frame. This is specified in the format sections of each class specification. The second bit in both classes is a sequence number. In the windowed info frame this is indicated by the word frame_id. This is necessary so that the protocol compiler knows how each frame is to be identified as it passes through the window mechanism.

In the info frame there is a field called info, which is the data portion of the frame. The maximum length of this field is specified, in this case ten, but the field may be assigned values of any length up to this maximum.

For each frame type there are two actions: a receive action and a send action. The receive action is executed after that frame type is received, and the send action is executed before it is sent. The actions are constructed using familiar high-level language constructs. The statements which resemble Pascal procedure calls are invocations of primitive actions defined on internal data structures. The retran primitive retransmits a frame from the send window, and the cancel primitive deletes an element from that window. The accept primitive in the receive action for info, sets the accepted flag in the receive window for the frame whose arrival caused this action to be executed. The receive window management system will pass the data to the layer above in the correct sequence at a later stage. The inc primitive may be used to increment state variables of the type field. The assignment to the data portion of the info frame is from the predefined variable called "data" which contains bits from the

layer above.

The final part of the specification is the timeout action. This is executed when a timer count expires. The timeout action in this case specifies that a single frame, the one causing the timeout is to be retransmitted. There is an optional section that is not used in this example. It is used to specify the action to be taken if the user-initiated timer expires. This will be discussed later.

It is important to note that while the scope of state variables is global, the scope of fields within class and frame structures is limited to the class or frame in which it is defined. A full specification of the syntax of PSL/1 can be found in the appendix.

To illustrate how PSL/1 can be used to specify the various protocols, a series of examples will be presented. The development of these examples will parallel the discussion of data link layer protocols in Tanenbaum(1981). Tanenbaum uses a series of examples written in an extended form of Pascal.

The first of these examples, in figure 3.3, is a positive acknowledgement / retransmission protocol. Each information frame is acknowledged by a single zero bit. Figure 3.4 shows a 1 bit sliding window protocol with piggy-backing. In a situation where data is flowing in both directions information and acknowledgement frames can be combined. The ack is said to ride piggy-back on the data frame. The protocol in Figure 3.5. introduces the concept of pipelining. This allows multiple outstanding frames. The "go back n" approach is adopted for retransmission.

The final example in Figure 3.6 illustrates the "selective reject" approach to retransmission. It also uses the user-initiated timer, which was mentioned earlier, to ensure data flow in one direction is not held up as a result of there being no flow in the other direction. After each data frame is received the timer is started using the start_timer primative. This timer is stopped as soon as a frame is sent to the other entity using the stop_timer primative. If there is no traffic in that direction for the specified timer_interval, the timer will expire and an

acknowledgement is sent. Notice also that a third type of action called a retransmission, or retran, action is used to ensure any retransmitted frame contains the sequence number of the most recent info frame accepted by this side of the protocol and not the last frame accepted when the frame was originally sent.

3.4. SUMMARY

An alternative approach to protocol specification has been presented together with some examples of its application. This approach was used in the development of the protocol modeling system that will be discussed in the next chapter.

```
protocol par

        /* A Positive Acknowledgement/Retransmission protocol */
        /* Tanenbaum protocol 3 page 147                      */

parameters {
        send_window:=1;          /* size of send window */
        receive_window:=1;       /* size of receive window */
        retran_interval:=10;     /* retransmission interval */
}

state{
        NextFrameToSend:="0";
        FrameExpected:="0";
}

class control direct{
        format{ "0"; }
        frame ack{
                action(receive){
                        if check_sum_error
                        then
                                retran;
                        else
                                cancel;
                        fi;
                }
        }
}

class info windowed {
        format{
                "1";
                seq[1] frame_id;
                info[10];
        }
        frame info{
                action(receive){
                        if check_sum_error
                        then
                                /* do nothing */
                        else
                                if seq=FrameExpected
                                then
                                        accept;
                                        inc(FrameExpected);
                                fi;
                                send(ack);
                        fi;
                }
                action(send){
                        seq:=NextFrameToSend;
                        inc(NextFrameToSend);
                        info:=data;
                }
        }
}

ontimeout{
        single_retran;
}
```

FIGURE 3.3 - A POSITIVE ACKNOWLEDGEMENT RETRANSMISSION PROTOCOL

```
protocol onebit_window

        /* A 1-bit sliding window protocol with piggybacking */
        /* Tanenbaum Protocol 4 page 152                     */

parameters {
        send_window:=1;          /* size of send window */
        receive_window:=1;       /* size of receive window */
        retran_interval:=10;     /* retransmission interval */
}

state {
        NextFrameToSend:="0";
        FrameExpected:="0";
        LastFrameAccepted:="1"; /* = 1 - FrameExpected */
}

class info windowed{
        format{
                seq[1] frame_id;
                ack[1];
                inf[10];
        }
        frame info{
                action(receive){
                        if check_sum_error
                        then
                                /* do nothing */
                        else
                                if seq=FrameExpected
                                then
                                        accept;
                                        LastFrameAccepted:=seq;
                                        inc(FrameExpected);
                                fi;
                                if ack=NextFrameToSend
                                then
                                        cancel;
                                        inc(NextFrameToSend);
                                fi;
                        }
                action(send){
                        seq:=NextFrameToSend;
                        ack:=LastFrameAccepted;
                        inf:=data;
                }
                action(retran){
                        ack:=LastFrameAccepted;
                }
        }
}

on_timeout{
        single_retran;
}
```

FIGURE 3.4 - A ONE BIT SLIDING WINDOW PROTOCOL

```
protocol pipelining

        /* Sliding window protocol with pipelining, */
        /* allows multiple outstanding frames       */
        /* Tanenbaum Protocol 5 page 158-159        */

parameters {
        send_window:=4;          /* size of send window */
        receive_window:=1;       /* size of receive window */
        retran_interval:=10;     /* retransmission interval */
}

state {
        NextFrameToSend:="00";
        FrameExpected:="00";
        LastFrameAccepted:="11";
        AckExpected:="00";
}

class info windowed{
        format{
                seq[2] frame_id;
                ack[2];
                inf[10];
        }
        frame info{
                action(receive){
                        if check_sum_error
                        then
                                /* do nothing */
                        else
                                if seq=FrameExpected
                                then
                                        accept;
                                        LastFrameAccepted:=seq;
                                        inc(FrameExpected);
                                fi;
                                cancel(AckExpected,ack);
                                AckExpected:=ack;
                                inc(AckExpected);
                        }
                action(send){
                        seq:=NextFrameToSend;
                        inc(NextFrameToSend);
                        ack:=LastFrameAccepted;
                        inf:=data;
                }
                action(retran){
                        ack:=LastFrameAccepted;
                }
        }
}

on_timeout{
        multiple_retran;
}
```

FIGURE 3.5 - PIPELINING

```
protocol nonseq_recv

        /* Nonsequential receive protocol - frames   */
        /*  - frames can be accepted out of sequence */
        /* Tanenbaum Protocol 6 page 162-163         */

parameters {
        send_window:=2;          /* size of send window */
        receive_window:=2;       /* size of receive window */
        retran_interval:=10;     /* retransmission interval */
        timer_interval:=5;       /* timer interval */
}

state {
        NextFrameToSend:="00";
        FrameExpected:="00";
        LastFrameAccepted:="11";
        AckExpected :="00";
        NoNak:=1;
}

class control direct{
        format{
                "0";
                kind[1];
                ack[2];
        }
        frame ack{
                kind=format{"0";}
                action(receive){
                        if check_sum_error
                        then
                                /* do nothing */
                        else
                                cancel(AckExpected,ack);
                                AckExpected:=ack;
                                inc(AckExpected);
                        fi;
                }
                action(send){
                        ack:=LastFrameAccepted;
                        stop_timer;
                }
        frame nak{
                kind=format{"1";}
                action(receive){
                        if check_sum_error
                        then
                                /* do nothing */
                        else
                                cancel(AckExpected,ack);
                                AckExpected:=ack;
                                inc(AckExpected);
                                retran(AckExpected);
                        fi;
                }
                action(send){
                        NoNak=0;
                        ack:=LastFrameAccepted;
                        stop_timer;
        }
}

class info windowed {
        format{
                "1";
                seq[2] frame_id;
                ack[2];
                inf[10];
        }
        frame info{
                action(receive){
                        if check_sum_error
                        then
                                if NoNak=1;
                                then
```

FIGURE 3.6 - A NON-SEQUENTIAL RECEIVE PROTOCOL

```
                                        send(nak);
                                fi;
                        else
                                cancel(AckExpected,ack);
                                AckExpected:=ack;
                                inc(AckExpected);
                                if seq=FrameExpected
                                then
                                        accept;
                                        NoNak:=1;
                                        LastFrameAccepted:=seq;
                                        inc(FrameExpected);
                                        start_timer;
                                else
                                        if NoNak=1;
                                        then
                                                send(nak);
                                        fi;
                                fi;
                        fi;
                }
                action(send){
                        seq:=NextFrameToSend;
                        inc(NextFrameToSend);
                        ack:=LastFrameAccepted;
                        inf:=data;
                        stop_timer;
                }
                action(retran){
                        ack:=LastFrameAccepted;
                        stop_timer;
                }
        }
}

on_timeout{
        single_retran;
}

on_timer_expired{
        send(ack);
}
```

FIGURE 3.6 - A NON-SEQUENTIAL RECEIVE PROTOCOL (Cont.)

CHAPTER FOUR

A PROTOCOL MODELING SYSTEM

4.1. INTRODUCTION

Given a physical connection between two computers, it will be possible to design several different protocols which will satisfy the basic requirement for an error-free communication path. Choosing a particular design will require some measure of the efficiency of each protocol. The effective transfer rate is one such measure. This measures the speed at which data is transferred across the link between the two machines, taking into account retransmissions due to errors and delays waiting for acknowledgements.

Traditionally estimates of protocol performance have been derived using traffic and queuing theory. Two examples of this approach are Field(1976) and Fraser(1977). Reiser(1982) is a comprehensive survey of this and other methods. More recently research has been conducted into predicting performance directly from formal protocol specifications. This can be done via simulation. Bauerfield(1982) discusses two formalisms which contain enough information for a simulation model to be automatically generated. One is a graphical representation called Function Nets which are related to Petri Nets, while the other is a high-level language called Hybrid Model.

Work was undertaken to show that PSL/1 could be used to generate simulations for protocol performance prediction.

4.2. DESCRIPTION

This work was conducted on the Departmental Vax 11/750 running the UNIX operating system. The model took the form of a system of communicating processes, with both protocol entities and communication channels being represented by individual processes.

The channel model was directly written in the C programming language, while the entity models were generated into C from PSL/1 specifications. At run time the complete system is produced from a single channel model using the fork and exec system calls. The single initial process spawned the entity models and finally forked itself to produce a full-duplex transmission model. This sequence of events is illustrated in Figure 4.1.

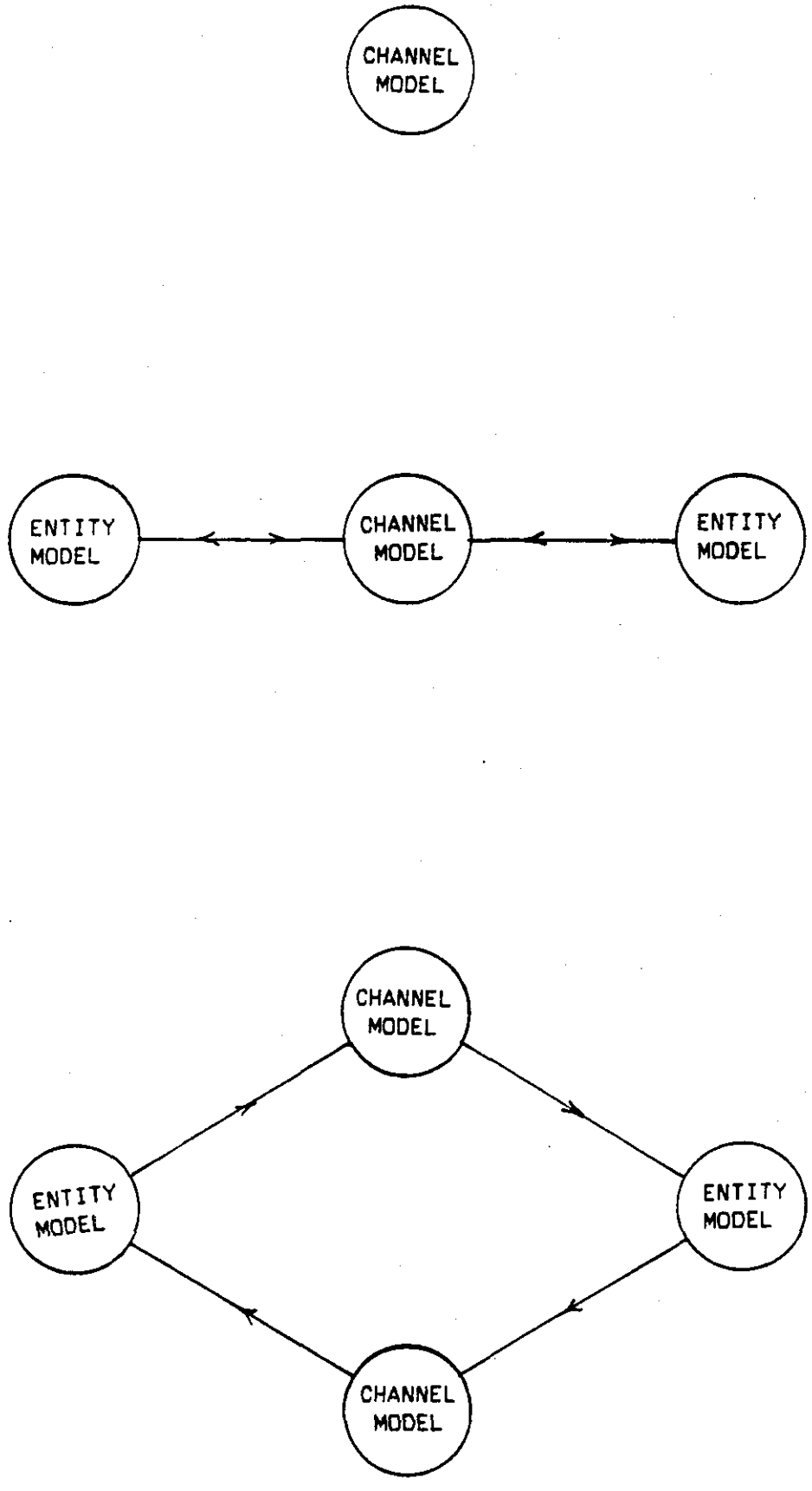


FIGURE 4.1 - PRODUCTION OF THE PROTOCOL MODELING SYSTEM

Slight modifications to the original design for PSL/1 were required. The parameters section of PSL/1 specification was reduced in size so that timer intervals could be specified at run time. In addition, the length of the data portion of the information frame could also be varied without recompiling the specification.

Three connection characteristics could also be specified at run-time. These were entity-to-entity propagation delay, line speed and bit error probability. These parameters could be varied from run to run to investigate their relationship with overall efficiency of a given protocol.

4.3. GENERATING AN ENTITY MODEL

The availability of compiler writing tools under UNIX eased the task of producing a PSL/1 to C translator. These tools are called YACC and LEX. YACC stands for "Yet Another Compiler-Compiler" (Johnson,1978b). It is a parser generator accepting specifications written in a grammar notation with embedded actions written in C. LEX (Lesk,1978) is a lexical analyer generator in many ways similar to YACC, but accepting regular expressions together with actions. With these it was possible to build a one pass translator for PSL/1.

The basic strategy taken was to generate five data structures from the specification.

(1) The symbol table

This is a linked list of elements containing a record structure with the following fields.

a) Variable name.
b) Variable type.
c) **if** b) = field **then** field length **else** zero **fi**.
d) An initial value.

(2) Class and frame definitions.

These are complicated structures of linked lists. At the top level we have a linked list of classes. For each class there is a list of frames in that class, and a linked list of

fields. For each frame in a class there may be a linked list of field redefinitions, each consisting of a list of fields. The field redefinitions will have a pointer to the field in the class definition they are redefining. This is illustrated in figure 4.2. These linked lists contain all the details required to manipulate the frame structures.

(3) A text file containing the send frame routine in C. This takes the form of a switch statement with a case for each frame type.

(4) A text file containing the receive frame routine again in C. This takes the from of a switch statement with a case for each frame type.

(5) A structure containing various miscellaneous details.

When the specification has been parsed and these five data structures are complete the output program can be produced. The initial C declarations are written to a file using information from the symbol table, frame and class definitions and miscellaneous details. Following this the main procedure is written. The send and receive routines are then appended to this file. A number of procedures were written to implement primative actions. These had to be combined with the output from the translator to produce an entity model which could then be compiled into executable code.

4.4. CONCLUSION

The approach presented above was successfully used to generate simulation models for a large variety of protocols similar to those presented in the last chapter. Unfortunately, restrictions imposed by UNIX made simulations very slow. This was because timer intervals could only be specified in seconds using the alarm system call. Therefore other timing, such as propagation delay, had also to be expressed on the same scale. Thus simulating a large data transfer would be very slow indeed.

Results from simulations which were carried out proved to be fairly erratic. In order to assess the affect of differing frame sizes a number of simulations were conducted. Each simulation

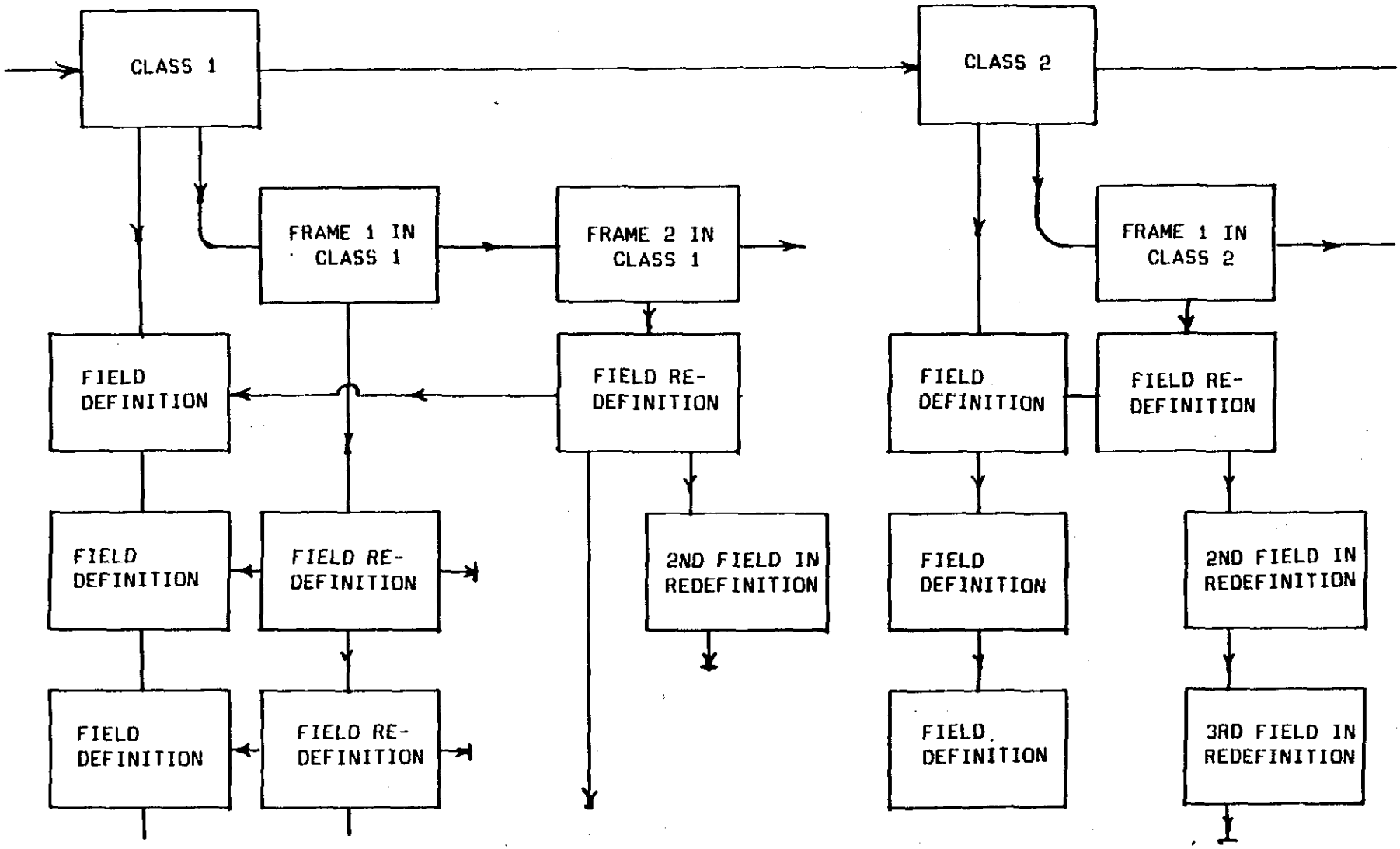


FIGURE 4.2 - CLASS AND FRAME DEFINITIONS

consisted of a transfer of 50,000 bits. The error rate was set at 10^{-4} and the propagation delay was set to five. The line speed was set to 9600 baud. The test was repeated 20 times for each frame size and an average taken. The results are summarised in figure 4.3. The results fail to show any clear trend when the frame size is greater than 4000 bits. The standard deviation within the set of tests for each frame size increased as the frame size increased. This last observation is predictable as a single error will have a greater impact on a transfer when the frame size is high.

However, PSL/1 had been shown to be a practical specification language for protocol simulation. The resulting simulation also proved to be a very useful tool for debugging protocol specifications. This was done by printing a message to a trace file every time a frame was sent or received. By increasing the error rate the protocol could be tested under extreme conditions and deadlock situations identified. As a consequence of this encouraging result, work began to apply this specification method to other areas of protocol design and implementation.

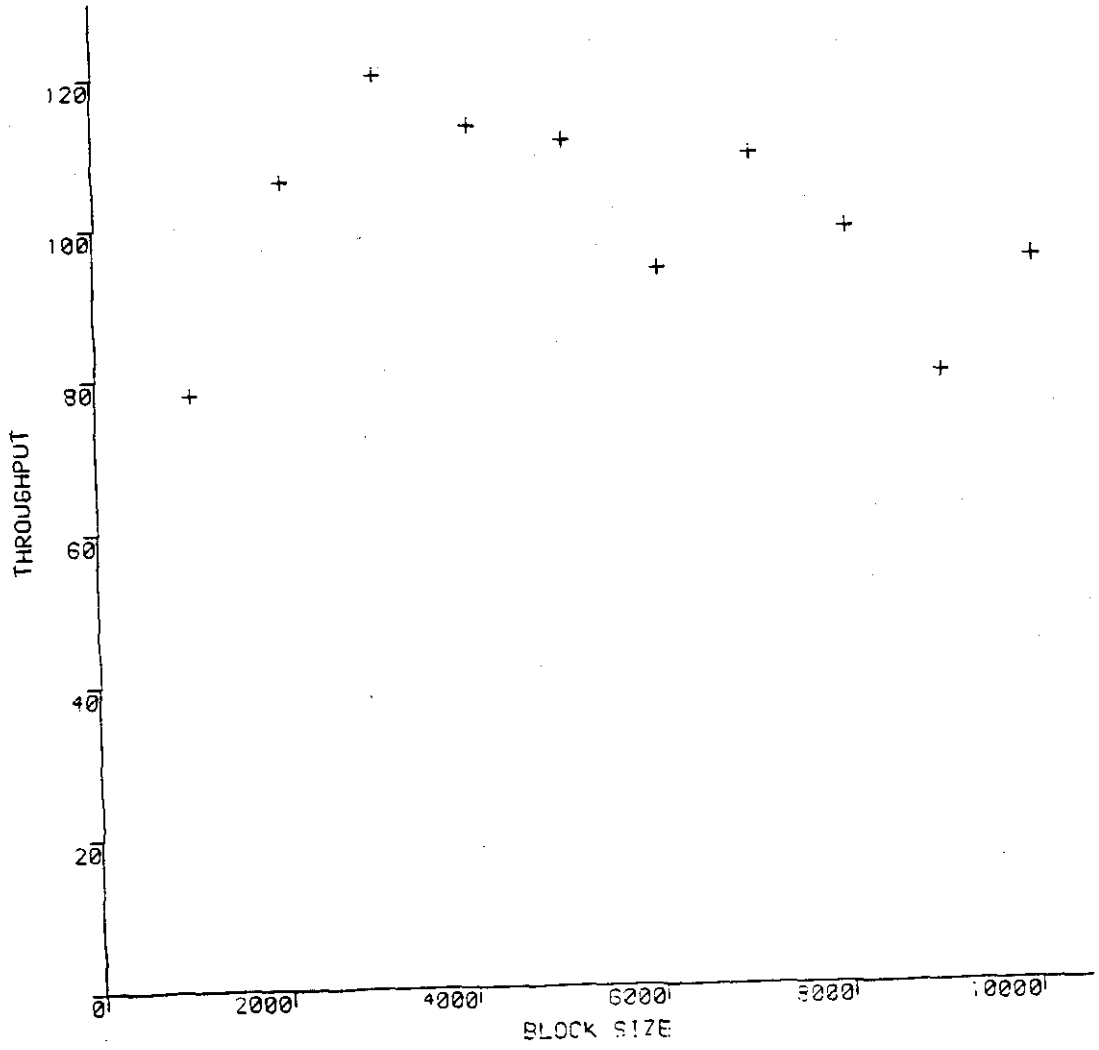


FIGURE 4.3 - SIMULATION RESULTS

CHAPTER FIVE

NETWORKING USING ASYNCHRONOUS INTERCONNECTION

5.1. INTRODUCTION

The work discussed in the previous two chapters was bit-orientated rather than byte-orientated. This is the approach that was adopted in most recent networking standards such as X.25, DECNET and SNA. Older networks such as ARPANET used a byte-orientated approach. The general adoption of bit-orientated protocols has been due to the desire to make network standards independent of any particular byte or word structure.

This approach is feasible where expensive networking equipment is available. However, users who only need communication facilities occasionally cannot justify such expenditure. Some computers, particularly microcomputers, can not be directly connected to a network. Hence, there is a need for a simple and cheap method of interconnection. The most readily available method is asynchronous character transmission via the ubiquitous V.24 interface.

5.2. NETWORK TOPOLOGIES

Asynchronous connection can be achieved in several ways. The simplest method is by linking each machine to every other by using an appropriately wired cable connection. However, this is only a practical solution where the number of machines(n) to be interconnected is small. Each machine will require n-1 ports dedicated to network traffic and a total of n(n-1)/2 cables. Where three machine are to be connected together the network will take up two ports on each machine and three cables in total. If the network grows to involve four machines, three ports will be required on each plus six cables. Five machines will require four ports on each machine and ten cables. At this stage the network is already consuming a significant quantity of resources. Hence, to conserve ports for terminal use and reduce the amount of cable required there needs to be some sharing of resources by the machines.

Many sites with multiple computer systems will already use a circuit switch to allow individual terminals to be connected to a different machine in each terminal session. Such a switch can also be used to allow computers to share ports and lines with each other and also with terminals. Each machine can be connected to a

switch as if it were a terminal. Hence each machine can login to any other machine providing a line from the switch to that machine is available. This situation is illustrated in figure 5.1. Once this has been done file transfers can be initiated between processes running on each machine.

An alternative approach is to use a local area network with an RS-232 asynchronous interface. An example of such a system is a low cost local area network called Clearway (Bidmead,1982 & RTDL,1984).

5.3. CLEARWAY

A Clearway system consists of several access units, or nodes, daisy-chained together into a ring. Each unit has an address in the range 1-99 and can be configured to initiate calls to other nodes. Alternatively, it can be configured to receive calls from an other node. Thus the roles of master and slave can be assigned to each node as required. One possible use of this system is to allow computers, particularly micro-computers to share resources such as printers. In this type of system a node attached to a computer will be permanently configured as a master node while a node attached to the printer will be permanently configured as a slave. Terminals may also be connected directly into a network via a node. Thus a Clearway network may be used in a similar way to a circuit switch, allowing an individual terminal to access more than one machine. In this case the nodes on the computer will be configured as slaves.

Under some operating systems it may be possible to use a node connected to a V.24 terminal port on a computer for incoming and outgoing connections at different times. In the normal situation the node is configured as a slave and the port is treated as an ordinary terminal port by the computer. In this situation, a terminal driver handles incoming connections. When the port is required for outgoing connections and there is no current incoming connection, the terminal driver can be disabled. The node can then be reconfigured as a master by the computer and an outgoing connection made. When the outgoing connection is no longer required the node can be restored to is original configuration and

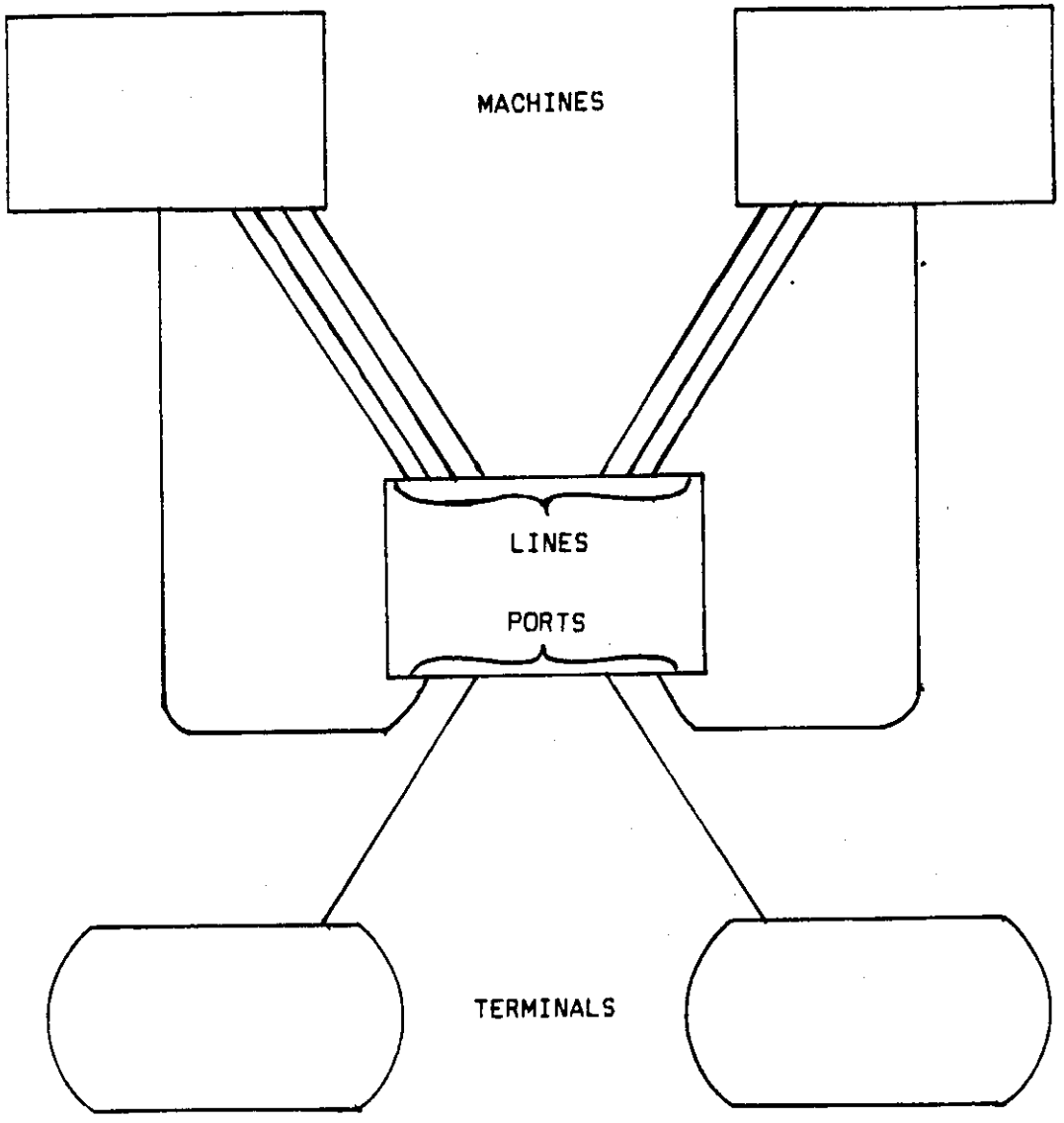


FIGURE 5.1 - USING A CIRCUIT SWITCH FOR NETWORKING

the terminal driver re-enabled.

Invisible to the user of Clearway there is a packet protocol operating between nodes. The data field of the packet can be up to 33 bytes long. Other fields contain sender and recipient identification, a sequence number, packet size indicator and a check character. After each packet is transmitted the sender waits for an acknowledgement before transmitting the next packet.

The ring speed is around 4500 characters per second (50K baud). This is slow compared to most local area networks. However cost factors must be considered. An Ethernet node costs between 250 and 300 pounds, and an interface to connect a computer to an Ethernet using an RS-232 interface will cost in excess of 1000 pounds. A parallel interface such as a DEUNA board for a VAX would cost at least three times that amount. On the other hand a Clearway node costs around 175 pounds. These figures have been enough to ensure wide spread use of this system.

With such a system one might argue it would be possible to send data at relatively high-speed, say 9600 baud, between connected computers. However, the RS-232 interfaces can be a problem area. The hardware of many computers has not been designed to coped with large amounts of high-speed incoming traffic such as that generated by a file transfer. Although they can receive data at that speed, there may be insufficient buffering or an inadequate interrupt handling system and hence they are unable to cope with an uninterrupted stream of bytes. With multi-user systems the computers ability to handle traffic may fluctuate depending on the load being placed upon it. With some computers XON-XOFF can be used across the RS-232 interface to increase reliability. However, some hosts do not support this type of flow control, or can not be relied upon to operate it without error. In these circumstances it is likely that characters will be lost by the recipient. Thus it is important to provide a layer of host to host protocol on top of that employed by Clearway to ensure reliable transfer of information.

It was decided that a Clearway network would be set up within the Department of Computer Studies at Loughborough University.

Initially, two mini-computers would be involved in the project, a Digital VAX 11/750 running UNIX and a system based on Texas Instruments 990/10s. The latter machine is a multi-processor system with four processors. In addition, some terminals in staff members offices would be attached to nodes allowing them access to use both the VAX and the TEXAS machines. The network would be required to handle both terminal access and file transfers.

The TEXAS machine was found to have particularly poor communication facilities. A program was written on the TEXAS machine to send a packet of data of variable length around the ring and read it back, comparing what was received with what was sent. This was tested at 9600 baud and it was discovered that the link became unreliable when the packet size became greater than ten. The TEXAS machine, therefore, requires an unusually short frame length if it is to receive the data intact. Reliability would also vary with the load on the machine, so a suitable protocol had to be found to ensure that data transfers could be achieved without loss of data.

5.4. *PROTOCOL STANDARDS*

Owing to the lack of suitable standards for communication over asynchronous links most systems of this type have been ad hoc responses to local needs. Often they have lacked even the most rudimentary error detection and recovery. They have also had a limited range of applications (eg file transfer only) and have only allowed interconnection of a small range of machines.

There is a pressing need for a standard for asynchronous networking. As yet no such standard has emerged, but two responses to the need are worthy of examination. These responses are the Kermit Protocol produced at Columbia University, New York (da Cruz,1983) and the proposals of the Transport Service Implementors Group of the British Telecom New Networks Technical Forum (BT,83) for an Asynchronous Transport Service (ATS).

The Kermit Protocol has been implemented on several different multi-user environments and on a large variety of personal computers. The user runs the Kermit on his local machine, connects to the remote machine and logs in. He then initiates the Kermit on

the remote machine and escapes back to the local machine by typing an escape sequence. At this point file transfers can begin between the two machines. When he has completed his transfers he must reconnect to the remote machine to logout.

The ATS proposals attempt to integrate asynchronous transfers into the ISO OSI framework. The proposals cover the data link, network and transport layers. They are based on the UK interim standard "Yellow book" Transport Service. At the lowest level a byte-orientated approach must be adopted due to the nature of the underlying communication channel, but above that level they try to follow ISO and UK standards as closely as possible.

5.5. FRAME REPRESENTATION

Simply adopting a byte-orientated approach is not sufficient to ensure correct transmission of characters. In many cases the hardware or software of the basic link does not allow transmission of all possible 8 bit codes. Some systems demand parity of some description or particular framing characters, such as Carriage Return or ETX, to ensure forwarding of the input from a front-end processor to a main frame or possibly low-level flow control such as XON-XOFF.

If the data to be transmitted only consists of 7-bit ASCII characters it is a fairly simple matter to code the control characters (octal 0-37) and the DEL character (octal 177) using escape sequences. The Kermit Protocol uses a special "quote" character to indicate that the next character is a coded control character. This coded control character can be decoded (and encoded) by exclusively ORing it with octal 100. The quote character can be any character in the range octal 41-76 and 140-176, although ` is the default value. A quote character is transmitted by preceding it by another quote character. If full eight-bit transmission is required another different quote character (default &) must be introduced to indicate that the following character, which may an encoded control character, has the eighth bit set. Thus up to three characters may be needed to transmit a single byte.

The ATS proposals offer a choice of two very different approaches. The first called transparent framing can only be used

in situations where the full range of eight-bit patterns can be transmitted. Each frame is preceded by a 3-byte header sequence.

Byte 1: octal 20
Byte 2: octal 202
Byte 3: length in bytes of the frame, including checksum.

The first two bytes were chosen since together they violate all four standard conventions for the eighth-bit (odd, even, mark, space). This may help to improve error recovery after loss or damaged characters has caused loss of frame boundary synchronisation. However, this is only true if text is being transmitted with consistent treatment of the parity bit. There is no way of improving error recovery if binary data is being transmitted.

ATS also provides another method of frame representation called hex-coded framing for links which are not fully transparent. The full eight bits are coded as two characters in the ranges 0-9 and A-F which together represent the hexadecimal value of the byte we are transmitting. In addition QZ,JZ,QX and JX are used as frame markers. This method should work for most connections since it does not require the ability to send or receive any non-alphanumeric characters. The specification allows for link-dependent variations whichever framing technique is used.

The Clearway network is not suitable for Transparent Framing since at least one character must not be sent across the network. This is the reset character which will put the node on the sender's side into configuration mode. It would be possible to implement Hex-coding Framing, but since two bytes need to be sent for each character we wish to transmit, this is very inefficient.

A frame representation similar to that used in the Kermit protocol was chosen for the Clearway network. Control characters in the data are coded into two bytes. The first byte is the Data Link Escape character, DLE, and the second is the character we wish to send ORed with octal 0100. Each packet is introduced by a control character. This must not be either DLE or the reset character. This scheme was chosen to aid error recovery by ensuring that frame boundaries could be easily restored.

Three numeric fields must be transmitted: a sequence number, a byte count and a checksum. The sequence number was constrained to be in the range 0-63. It is increased by 64 before transmission to ensure it is not a control character. The byte count could require one or two bytes depending on the maximum length of the data field. Similarly, the checksum could be either two or three bytes long, depending on the frame size. Such flexibility was introduced because of the restrictions placed on the overall frame size by the Texas Instruments machine.

Where the length of the data field in a frame must be less than 64, a single byte is sufficient for the byte count. The seventh bit is set before transmission to exclude control characters. Two bytes must be used for frames with data fields longer than 64 bytes, but shorter than the maximum of 4095 bytes. The byte count is divided into two six-bit quantities which have the seventh bit set before transmission. The checksum is simply the sum of all the bytes in the frame. It is treated in a similar way to the byte count in that it is divided into six-bit quantities and converted to printable characters.

An optional terminator may be used to ensure forwarding of frames. This was included because the Berkeley Network Discipline requires a newline to terminate each message. This scheme was not designed to carry binary data.

5.6. PROCEDURAL ASPECTS

As has been previously mentioned the ATS proposals follow the ISO OSI approach as closely as possibly. The designers of the Kermit Protocol, however, were free to devise their own procedures. One feature of the Kermit approach is that each side can configure the other by informing it of its particular needs when transmissions are initiated. The Kermit protocol was designed to operate over both full and half duplex connections, but in a half duplex manner. Hence, Kermit is a send and wait protocol.

For the Clearway network it was felt that the possibility of a full sliding window protocol was required, but one that was simpler than X.25. In order that the individual characteristics of each pair of hosts could be used to allow the most efficient

protocol possible to be used for each connection, an initial exchange of information is required between hosts. This is similar to the Kermit Protocol. In the protocol for the Clearway Network, the host which is to be the receiver must specify the maximum send window size and the maximum frame size. In addition it indicates whether it requires a terminating character on each frame it receives and, if this is so, which character is required.

5.7. SUMMARY

The special problems of networking over asynchronous line has been discussed, together with strategies for overcoming them. The Clearway networking system has been presented together with an outline of the protocol chosen to operate on it. The rest of this thesis will be concerned with how such a protocol can be implemented across a network in way that is both efficient and easy to maintain.

CHAPTER SIX

PROTOCOL IMPLEMENTATION

6.1. INTRODUCTION

By the time a protocol specification is approved a lot of hard work will have gone into the design and validation of the protocol. However, at this stage of development there has still been no change to the computers that will be involved in the network, which will still be operating an old protocol or isolated from each other. There is still a great deal of work to be done before the protocol is fully implemented.

How much work remains to be done will depend on the number of different types of computer involved. If the network involves machines of exactly the same type then there is only one implementation of the protocol required. If all the machines in a network are from the same manufacturers range it may not be too difficult to adapt the initial implementation to run on the other machines. However, many networks involve machines from a wide range of vendors, and this can result in a lot of extra work. The purpose of this chapter is to consider ways to implement protocols on this type of network avoiding excessive duplication of effort.

6.2. USE OF HIGH LEVEL LANGUAGES

If the process of producing an implementation on a new machine can be made fairly mechanical, the probability of introducing errors can be reduced. One way to do this is to use a high-level language which is available on a wide range of machines. The most common high-level language which might conceivable be used is Fortran. However, many versions of Fortran do not allow sufficient access to the operating system to make this feasible.

The two languages commonly used for system programming are PL/I and C. PL/I was devised by IBM, but it is also used by Honeywell in their Multics system. C is the language which is used to write much of UNIX. UNIX is a portable operating system, not tied to any particular manufacturer. It has been implemented on to a large number of machines, including DEC PDP and VAX machines, GEC computers, Perkin Elmer machines and a host of micros and work stations. The XENIX system produced by Microsoft is essentially the same operating system but on a smaller scale.

However, C is not restricted to UNIX, it is available under TOPS-20, VMS and also on IBM, Amdahl and Honeywell machines.

The different implementations of C are, however, not without system dependent characteristics. In fact, this is inevitable because of the access the language gives to the machine and operating system on which it is implemented. UNIX provides a large library of both source and object code which can be used for input/output and other common tasks. Some manufacturers provide an equivalent library which can be used with their particular C compiler. This is particularly true of VMS where UNIX system calls can be emulated using a set of routines with the same interface. Hence C programs can become highly portable.

Where differences are unavoidable, owing to differences between terminal drivers, the C macro-processor has a conditional compilation facility which can be used to allow different sections of code to be used on different machines. There is a C version of the Kermit Program which was written so that only a simple change is required before it will compile on one of the other supported machine. This change is simply to swap two characters in the source file. The source for this Kermit contains the following lines very close to the beginning of the text.

```
/* Conditional compilation for different machines */
/* and operating systems */
/* One and only one of the following should be 1 */

#define UCB4X   1 /* Berkeley 4.x UNIX */
#define TOPS_20 0 /* TOPS_20 */
#define IBM_UTS 0 /* Amdahl UTS on IBM systems */
#define VAX_VMS 0 /* VAX/VMS (not yet implemented) */
```

The position of the 1 in the four #define lines determines the operating system under which the program will be compiled. Code appropriate to a particular operating system can then be selected for compilation at certain places in the program. However, there still be problems with supposedly portable languages, like C, which relate to how they fit into a particular operating system. The alarm system call can be used to generate a signal after a

certain number of seconds. This can be used to timeout an entity that has not received an acknowledgement. The programmer writes a signal system call specifying a routine which will handle the signal. This is followed by an alarm call specifying the timeout interval. This will be followed by a read which waits for the acknowledgement. A signal results in the specified routine being called. Under UNIX, on returning from this routine the read system call terminates, returning with an error code. The signal handler can in this case be a do nothing function. The following sections of code illustrate this approach.

```
#define OK      0
#define TIMEOUT 1
#define SYSERROR -1
        :
alarmcatch(){}
        :
signal(SIGALRM,alarmcatch);
alarm(10);  /* 10 second timeout */
while(read(net,buf,sizeof(buf))==SYSERROR)
{
        retran(last_frame);
        signal(SIGALRM,alarmcatch);
        alarm(10);  /* 10 second timeout */
}
alarm(0); /* turn off alarm */
        :
```

However, under VMS when control returns from alarmcatch the system call continues to wait for input. The use of the setjmp and longjmp facilities from the standard libraries would appear at first to offer a way round this problem by avoiding the normal return. However, the C compiler release notes for VMS indicate that the setjmp and longjmp can not be used in this way. VMS appears to require the following technique.

```
alarmcatch()
{
        retran(last_frame);
        signal(SIGALRM,alarmcatch);
        alarm(10);  /* 10 second timeout */
}
        :
signal(SIGALRM,alarmcatch);
alarm(10);  /* 10 second timeout */
if(read(net,buf,sizeof(buf)==SYSERROR)
{
        /* we have a real system error ! */
}
alarm(0); /* turn off alarm */
```

This code will not work under UNIX. However, a small change will render it portable.

```
alarmcatch()
{
        retran(last_frame);
        signal(SIGALRM,alarmcatch);
        alarm(10);  /* 10 second timeout */
}
        :
signal(SIGALRM,alarmcatch);
alarm(10);  /* 10 second timeout */
while(read(net,buf,sizeof(buf)==SYSERROR);
alarm(0); /* turn off alarm */
```

As long as their are no real system errors this code will be alright. For safety a limit on the number of retries would need to be incorporated into the code.

An alternative approach would be to use conditional compilation to compile a different system call for VMS instead of the UNIX read.

A language like C may be useful if it is available on all the machines in the network. Sadly this is not usually the case. The

TEXAS machine on the departmental network had only an assembler and Fortran and Pascal compilers. Hence TEXAS assembler was chosen as the most suitable language for an implementation on that machine.

6.3. PROTOCOL COMPILER

Although it does not solve the portability problem, using a protocol compiler to implement a protocol in an automated fashion can greatly reduce the the amount of work involved. There have been two major pieces of work concerned with protocol compilers. One approach was used by IBM to allow its users to generate software for SNA, while another has been used on an early version of the ISO subgroup B language.

The work done by IBM is principally described in Nash(1983). The language used was FAPL (Format and Protocol Language), which was described in Chapter 2. The target language was PL/I, although several different dialects of PL/I could be produced. The compilers on each individual machine were used to produce executable code. The variations between dialects were obtained by writing the code generation phase in such a way that it uses a set of code generation macros. These could be varied to cater for each different dialect and systems environment. The macros have well defined interfaces and functions. Sample versions are supplied to the user who can the tailor them to his own particular requirements. The macros are written in REX, a PL/I-like general purpose language. There are about 40 such macros. There are some FAPL functions that can not easily be coded into the target language. In this case run-time support routines are used.

The basic principles used here represent a sound approach to the implementation of protocols on a range of machines. Its main weakness is that the approach has only been applied within a particular manufacturers range, and it has only been used to produce code in dialects of the same language.

The other work in this field was described in Blumer(1982). A protocol compiler was constructed for an early version of ESTELLE, using the YACC and LEX programs. The target language was Pascal. A finite state machine can be represented by a set of

tables which guide program execution depending on external events. The YACC and LEX systems are built using this principle. For YACC and LEX the external event is reading a character or token, and the appropriate action will be updating internal structure or outputting some code. This same principal can be applied to protocol programs. The external events in this case will include frame arrivals, the arrival of data from the layer above, and timeouts and the appropriate actions would include sending a frame and closing down a connection.

Hence, in this work the protocol compiler outputs a set of tables from the specification and adds code to traverse them. These tables are the same for all protocols. A set of actions is also produced from the specification. The latter can be done with the minimum of processing since the actions in the specification are already written in Pascal. These three items together can be compiled into a protocol program.

This work shares the same weakness as the work using FAPL in that it requires a Pascal compiler to be available on all the machines in a network. The portability problem has still not been tackled.

Neither of these approaches is sufficient to ease the problem of implementing software on the Clearway network, since they have only tackled translation to a high-level language. There is a gap to be bridged from the high-level language to the assembler code. This gap is usually filled by a compiler. This suggests the possibility of using a portable or retargetable compiler to produce the final code.

6.4. PORTABLE AND RETARGETABLE COMPILERS

Suppose there is a compiler for language A operating on machine X, which we wish to move to machine Y. Further suppose that this compiler is written in a language B for which there is a compiler on machine Y. We can transfer the source code for the compiler from machine X to machine Y, compile it, and we should have a compiler that accepts language A on machine Y. Unfortunately, it will still produce code for machine X. Hence the source for this compiler will need to be altered to generate code

for machine Y. How simply this can be done will depend on the internal structure of the compiler. A compiler that has been designed so it can easily be moved to another computer and adapted to generate a different target code is known as a portable compiler.

A retargetable compiler is essentially the same except that it is not intended to move the compiler onto a new machine but generate code for another machine on the original machine. A retargetable compiler can be used where there is not a compiler for language B on machine Y. In some cases it may not even be possible for machine Y to support any compilers owing to limitations on memory space. Thus a program can be compiled on a mainframe or mini-computer and down line loaded on to a small microprocessor.

Compilers of this type need to be structured in such a way that it is easy to adapt the code-generation phase to a new machine. There needs to be a clear separation between the language dependent and machine dependent parts of the compiler.

One approach has been to produce code-generator generators(CGG) in a similar way to compiler-compilers such as YACC and LEX. Some type of specification language is accepted by the CGG to produce code for the code generation phase of a compiler. The specification language can take one of two forms:

(1) A specification of the target machine and its instruction set.

(2) A specification of the translation process between some form of intermediate code and the target assembler.

The first approach would be preferable if some form of standard machine description was provided with each machine. However, a standard machine description language has not been adopted and devising such a language would be a major task in itself. This approach is discussed in Cattell(1980) and was found to be very complex.

The second approach requires that the user knows enough about his particular machine to see how intermediate code concepts can

be translated into machine code. This approach is described in Granville(1978). In this work, a series of code templates, similar to the macros used in the FAPL work, were used as the basis for the specification. Associated with the templates were coded instructions indicating were each template was to be applied. The intermediate form used as the starting point can be either an intermediate language or a code tree. Granville(1978) uses an algebraic notation as the specification language. The C portable compiler (Johnson,1978a) uses a code tree as its starting point.

Poole(1974), Colman(1974) and Waite(1970) describe a system based on a linear intermediate code. Poole(1974) describes the concept of abstract machine modelling which underlies the work in these three papers. An abstract machine is a generalised machine architecture designed to be a common sub-set of as wide a range of machines as possible. A family of abstract machines called JANUS was devised and a well structured abstract machine code defined.

The basic approach adopted was to clearly divide the compiler into language and machine dependent parts. The former they called the language dependent translator (LDT), and the latter they called the machine dependent translator (MDT).

The LDT contains all the lexical and syntactic analysis necessary for the particular source language. If a program is parsed successfully the compiler will determine what actions are required to execute the program and then pass a specification of these actions to the MDT. To keep the LDT machine-independent the actions it produces must not rely on a particular target computer; they must be fundamental operations which can be implemented on any computer. The MDT must translate these operations into the assembly code for a particular machine. The information flow from the LDT to the MDT is in the form of abstract machine code. A program called STAGE2 (Waite,1970) was used as the MDT. This was driven by a set of translation rules supplied by the user.

The data types of JANUS are high-level entities such as integers, addresses and real numbers. The operations on the other hand were the lowest form possible, such as load a, add b, etc.

6.5. CONCLUSION

Three areas have been investigated in the search for techniques to simplify protocol implementation. The use of high-level languages, despite possible pitfalls, was shown to be a useful approach. A protocol compiler would reduce the effort required to convert the specification into a suitable high-level language. A portable compiler could be used to implement the high-level language chosen.

An obvious course of action was to retarget the existing portable C compiler to produce TEXAS assembler. However, experience within the department had shown that this was a lengthy process which would probably not be cost-effective. TEXAS assembler has a fairly asymmetric instruction set which might have proved difficult to mould into a DEC-orientated code generation system.

Hence, an alternative approach was chosen. This was to produce a retargetable protocol compiler on the VAX which would produce code for a variety of machine types.

CHAPTER SEVEN

A RETARGETABLE PROTOCOL COMPILER

7.1. INTRODUCTION

The discussion in the preceding chapters has examined various aspects of protocol specification and implementation. The problems that can arise implementing protocol entities on a range of machines have been presented and the last chapter suggested that a retargetable compiler might represent a step forward in this area. The compiler would adopt the approach described in section 6.4. Hence an abstract machine would be used as an interface between language dependent and machine dependent parts of a compiler. The task of producing a retargetable compiler can be divided into several steps.

(1) PROTOCOL LANGUAGE REVISION

There were several reasons for doing this. Firstly, PSL/1 had been developed for bit-orientated protocol whereas byte-orientated protocols were now required. Owing to this change of direction, the interfaces with the layers above and below had to be redesigned. Experience had shown that the two-tier frame declaration system employed in PSL/1 was unwieldy and unnecessarily complicated since only simple protocols were required. There were several other changes made to the protocol specification language which will be described later. The resulting language was called PSL/2.

(2) ABSTRACT MACHINE DESIGN

This step consisted of designing an abstract machine together with an associated assembly code. The JANUS abstract machine could be taken as a starting point. However, this machine was devised without taking into consideration microprocessor architectures so the basic structure of the abstract machine required some modification.

(3) LANGUAGE DEPENDENT TRANSLATOR

A protocol compiler had to be written to translate protocol specifications written in PSL/2 into programs written in the assembly code of the abstract machine. The assembly code for the abstract machine is called I-code.

(4) MACHINE DEPENDENT TRANSLATOR

Another translator was required to produce the equivalent of

an abstract machine program in a wide range of assembler codes.

(5) DESIGN OF A MACHINE INDEPENDENT OPERATING SYSTEM INTERFACE
A set of interface routines are required for each different machine type to implement machine dependent aspects of the protocol entities.

The rest of this chapter will examine each of these steps in detail.

7.2. PSL/2

The switch to a byte-orientated approach made it necessary to make various changes to the interface with the layer above. In PSL/1 bit streams had been used to send data to the layer above and receive data from it. In PSL/2 these bit streams became byte streams.

In addition some requests and indications were defined on the interface with the layers above and below. The layer above and the layer below both require some means of indicating to this entity that a connection has been established with a peer entity. This is done by means of an OPEN_REQUEST. The entity does not establish the connection, so this must be done by another piece of software. The layer above requires some means of telling the protocol entity that data had been made available. The CHARACTER_ABOVE indication was defined for this purpose. The layer above also requires some means of instructing the layer below to close the transaction with the peer once all outstanding data has been transmitted and acknowledged. This was done by defining the CLOSE_REQUEST. If an entity receives frames which break the rules of the protocol it can communicate this fact to the layer above by sending an error indication. A primative action called ERROR is provided in the PSL/2 language for this purpose.

The interface with the layer below is similar to that which was used in PSL/1 except that frames are now made up of bytes not bits. The interface with the layers above and below are illustrated in Figure 7.1.

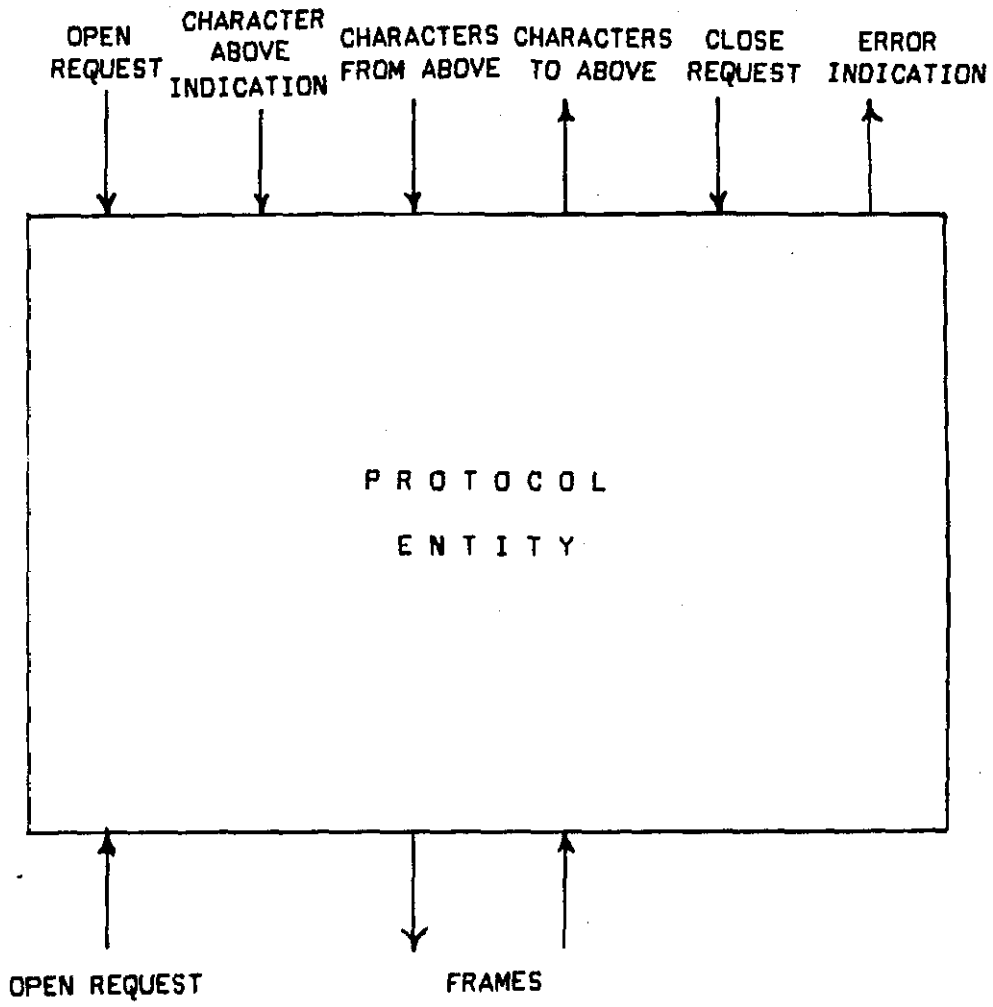


FIGURE 7.1 - A PROTOCOL ENTITY

An explicit state variable was introduced into PSL/2 which brought the language closer to existing extended finite state machine languages such as ESTELLE and FAPL. This variable is similar to the enumerated type found in C and Pascal. The specifier defines a limited set of values it can take in the form of alphanumeric names. The value of the state variable is changed using the NEW_STATE primative. Associated with each value there will be expectations regarding peer entity behaviour. These expectations will be reflected in the specification.

The two-tier frame declaration system was replaced with a record structure style of declaration. However, each frame is not described as a partitioned section of memory, but as a concatenation of constants and variables. This type of declaration can be viewed as a set of assembly and disassembly instructions for each frame, thereby eliminating the need for assignment statements.

The integer type was dropped, in favour of a type called FLAG. This is equivalent to a boolean variable in Pascal. The type called SEQ_FIELD replaced the FIELD type of PSL/1. This is a field of undefined length, but it was expected to be implemented as a single byte. The field ID_FIELD is a byte length field which has to appear at the beginning of a frame declaration and can appear nowhere else. It is used to identify a frame when it arrives.

In addition some purely cosmetic changes were made to the syntax. For example, some redundant characters such as semicolons and brackets were removed.

An example of a specification written in PSL/2 can be found in figure 7.2. It is incomplete, but it helps to illustrate the changes that were made. Some finite state machines illustrating the construction of the example are presented in figure 7.3. A full syntax of the language is given in the appendix.

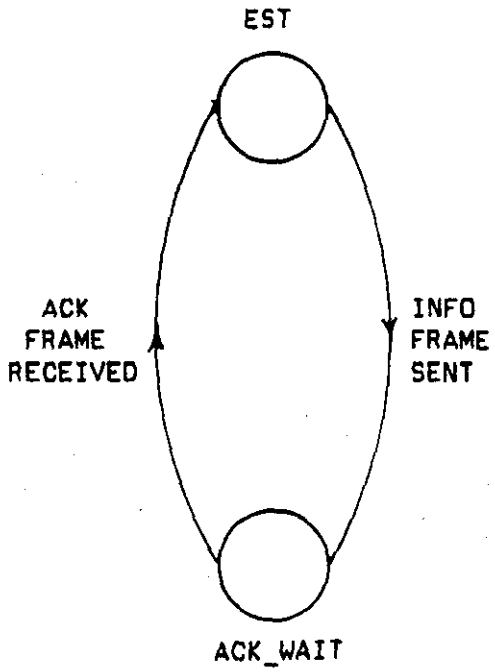
PSL/2 is not case sensitive, therefore upper-case can be used to highlight the reserved words of the language. This has been done in the example. The first line identifies the protocol, which is a positive acknowledgement retransmission protocol. Following this sequence number variables are declared. They are

```
PROTOCOL Par

SEQ_FIELD NextFrameToSend,LastFrameSent,ReceivedFrame

STATE estab,ack_wait

WINDOWED FRAME info
        ID_FIELD 'STX'
        SEQ_FIELD ON_RECEIPT ReceivedFrame ON_SEND NextFrameToSend
        DATA
        CHECK_SUM
END_FRAME

DIRECT FRAME ack
        ID_FIELD 'ACK'
        CHECK_SUM
END_FRAME

EVENTS
        ON_OPEN_REQUEST
                OPEN_R_WINDOW
                NEW_STATE estab
                DEC LastFrameSent
        ON_CHARACTER_ABOVE
                IF estab THEN
                        SEND_BELOW info
                        INC LastFrameSent
                        INC NextFrameToSend
                        IF S_WINDOW_FULL THEN
                                DISABLE_ABOVE
                        FI
                        NEW_STATE ack_wait
                ELSE
                        ERROR
                FI
        ON_CLOSE_REQUEST
        ON_CHARACTER_BELOW
        [info]:
                IF estab THEN
                        RECEIVE
```

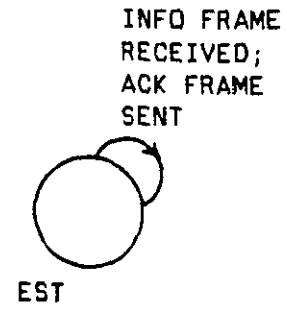
FIGURE 7.2 - PSL/2 EXAMPLE

```
                IF IN_R_WINDOW THEN
                        SEND_ABOVE
                FI
                SEND_BELOW ack
        ELSE
                DISCARD
        FI
    [ack]:
        IF ack_wait THEN
                RECEIVE
                CANCEL LastFrameSent LastFrameSent
                ENABLE_ABOVE
                NEW_STATE estab
        ELSE
                DISCARD
        FI
    ON_TIMER_EXPIRED
END_EVENTS
```

FIGURE 7.2 - PSL/2 EXAMPLE (cont.)



SENDER(S)



RECEIVER(R)

S U R =

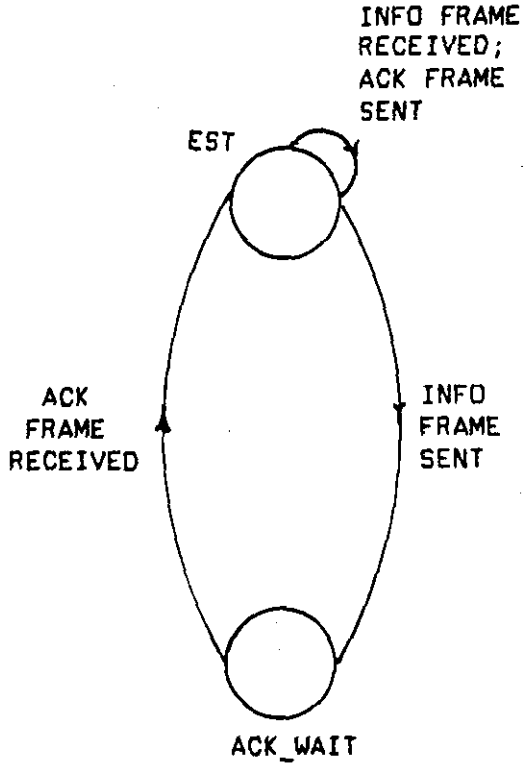


FIGURE 7.3 - FINITE STATE MACHINES FOR PSL/2 EXAMPLE

automatically set to zero at the start of each transaction. Next the state variable is declared. There are two possible values of this variable: estab and ack_wait.

The info frame is declared next. This is an example of the new style of frame declaration. Following the ID_FIELD there is a sequence number. When an info frame is sent the value for this field is obtained from the variable NextFrameToSend. Similarly, when an info frame is received the value of this field is placed in the variable ReceivedFrame. Following this field there is a DATA field. Finally there is a CHECKSUM.

The rest of the specification consists of a list of request and indication events and associated actions which are executed when these events occur. The ON_OPEN_REQUEST event occurs whenever the entity is invoked at the start of a transaction. This event occurs irrespective of whether the entity is to send or receive data. It is not necessary that there be any mechanism for informing the entity which role it is to take, since the arrival of data from above will automatically nominate the sender.

The next event is called ON_CHARACTER_ABOVE. This event occurs when a character is made available by the layer above. A SEND_BELOW operation within the action associated with this event will collect such characters until it reaches the maximum frame size or no character is made available for a specified interval.

The event ON_CHARACTER_BELOW occurs when a character is made available by the layer below. This should indicate the start of a frame from the peer entity. The character read is compared with the ID_FIELD of each frame type listed for the event. Each element in this list is enclosed in square brackets followed by a colon. If a match is found the action associated with that frame type is executed. If no match is found characters are read until a match is found or there are no more characters to read. An entity can disable the ON_CHARACTER_ABOVE event by using the DISABLE_ABOVE primative. When an entity is again ready to receive characters the ENABLE_ABOVE primative can be used.

The ON_TIMER_EXPIRED event has the same function as the event of the same name in PSL/1. There is no ON_TIMEOUT event, and only

single frame retransmission is supported. Multiple retransmission is achieved by repeated timeout on the sender side.

Many of the primitive actions of PSL/2 are equivalent to those of PSL/1 or are the same as PSL/1 primatives with a different name. The SEND_BELOW primitive in PSL/2 is equivalent to the SEND primative in PSL/1. Similarly, the SEND_ABOVE primitive in PSL/2 is the same as the PSL/1 ACCEPT primative. The CANCEL, RETRAN, START_TIMER, STOP_TIMER, INC and DEC primitives are exactly the same as in PSL/1.

There are some fundamental differences between PSL/1 and PSL/2 in the way frames received by an entity are processed. In PSL/1 frames were considered to be indivisible, whereas in PSL/2 they are treated as a sequence of characters. When a valid ID_FIELD is found there are two choices: the entity can either RECEIVE the frame, that is accept it, or it can DISCARD it and search for a new ID_FIELD value. A frame may be discarded if it arrives out of context. In the example if an ack is received when there are no info frames outstanding, it can probably be discarded.

The OPEN_R_WINDOW primative is used to initially set up the receive window. A FLAG is set to true using the SET primitive and it is set to false using the UNSET primitive.

The range of conditional expressions was extended in PSL/2 by the addition of OR, AND and NOT operators. Some special conditions were also added. S_WINDOW_FULL is true if the send window is open to its full extent. IN_R_WINDOW is true if the sequence number of the last frame received is within the receive window.

7.3. THE ABSTRACT MACHINE

The concept of an abstract machine was discussed in section 6.4. An abstract machine should have a common subset of the features of a wide variety of existing machines and acts as an interface between language and machine dependent parts of a compiler. Formulating such a machine is difficult due to the tremendous differences there are between different machines. For example, the number and characteristic of registers varies greatly

from machine to machine. In order to avoid problems in this area an abstract machine was devised which has no general registers. The instructions written in the abstract machine code may well require the use of registers on a real machine, but at this level no attempt is made to generalise.

There is, however, an index register that can be used to address array elements. The abstract machine has a stack which can be used for parameter passing. A set of condition codes were defined as part of the machine.
These codes are:

eq - equal,
ne - not equal,
lt - less than,
gt - greater than,
le - less than or equal,
ge - greater than or equal,
true,
false.

A symbolic assembly language is associated with this machine. This language is called I-code. The syntax of this language is described in the appendix. Four types of instruction are defined in I-code. They are:

(1) Allocation instructions.
These are either variable instructions for simple variables or array instructions for more complex structures.

(2) Arithmetic two operand instructions.
These include addition, subtraction and move instructions.

(3) Arithmetic one operand instructions.
These include instructions for incrementing variables and stack manipulation.

(4) Control instructions.
These include segment delimiters and branch instructions.

It was envisaged that at some point in the translation process I-code would need to be expressed in a very simple form.

Thus a five element tuple was designed for this purpose. Most instructions are represented by a single tuple, except arithmetic two operand instructions which require a pair of tuples. In each tuple the first element is either an operator or a zero. A zero indicates this is the second tuple of a pair. The remaining elements usually specify an operand of the instruction. If the instruction does not required an operand these elements are zero filled.

The second element is usually the type of the operand. The are four types defined:

char - character,
seq - sequence number,
addr - address,
int - integer.

(The type "sequence number" is used to represent the addressing unit used to store frame sequence numbers.) However, in jp instructions this second element is used as an optional qualifier to the operand.

The third element is the mode of access, which is used together with the next two elements to access the storage already defined in allocation instructions. The next element is an identifier and the last element is a constant integer value.

There are six possible modes: variable, inx_use, inx_offset, inx, param and const.

(1) The variable mode implies that the operand is found at the address associated with the identifier which follows it. The identifier will have been defined by a variable instruction.

(2) The inx_use mode implies that the value in the index register is used to determine the location of the operand.

(3) The inx_offset mode implies that both the value of the index register and the value of the final element of the tuple are added together to give the exact location of the operand. This offset is expressed in terms of bytes.

(4) The mode inx refers to the index register itself. This mode may only be used in a tuple in which the second element is set to addr.

(5) The param mode refers to parameters located on the stack. If this mode is used, the final element is used as an index into the stack. A value of 0 refers to the top of the stack, while a value of 1 the next element down, and so on.

(6) The final mode is the const mode. The value of the operand is contained in the last element of the tuple and there is no identifier in the fourth field.

Programming using five element tuples would be very tedious, hence, the symbolic code for the abstract machine is not written in this form. However, an I-code program should be seen as a convenient shorthand for a sequence of tuples. The I-code for each type of instruction will now be discussed in detail.

7.4. I-CODE

7.4.1. ALLOCATION INSTRUCTIONS

The most complex structures we want to manipulate are the send and receive windows. These are fixed length arrays of fixed length structures. Therefore, only two types of allocation instructions are needed: the variable instruction for simple variables and the array instructions for the more complex structures. A variable instruction specifies the type, name and initial value of a variable. The initial value may be omitted, in which case a default of zero is assumed. The number of bytes allocated can be calculated using the formula

bytes = len(type)

where the len function gives the number of bytes used to represent that type on a particular machine. The array instruction specifies the type, name and length in terms of the number of elements of an array. The number of bytes allocated can be calculated using the formula

bytes = len(type) x length of array.

7.4.2. ARITHMETIC TWO OPERAND INSTRUCTIONS

An arithmetic two operand instruction consists of an operator followed by a pair of operands separated by a comma. The format of the operand depends on the mode of access employed. If the variable mode is used, the operand will just consist of the identifier name. If the inx_use mode is used the operand is written as "[inx]", and if the inx_offset mode is used the operand is as "<offset>[inx]", where <offset> is an integer constant referring to a number of bytes. If the param mode is used the operand is written "<offset>[param]", where <offset> is an optional integer constant referring to a stack element number. If the offset is omitted the operand is at the top of the stack.

An operand with the const mode can be written in one of two ways. If the constant is a character it can be enclosed by a pair of single primes and the conversion to the ASCII character code is achieved during translation. Any other form of constant can be written as an integer. When the index register is used as an operand this a written as "inx".

In order to generate the correct assembler code from an i-code instruction it is necessary to know the type of each operand. For operands of the variable mode this is recorded in the variable instruction that declared it. The type of a character constant is also obvious from the form in which it is written. However, for other modes the type is not immediately obvious from any single operand. Normally, however, the type of an operand will be the same as that of the other operand in the instruction. Therefore, if the type of a operand is not known the type of the other operand is assumed. However, in some cases the other operand will also be of unknown type. This would be the case, for example, if a non-character constant was being assigned to a operand with inx_use mode. In such a case one or both of the operands must be cast. A cast precedes an operand and consists of the required type enclosed in parenthesis. For example:

```
(char) [inx]
```

Casts may be used at any time to overrule defaults.

There are five operators defined: mov, add, sub, cmp and gad. In each case the source operand is the right operand, while destination operand is on the left. The purpose of the first three operators can easily be deduced from their names. The mov operator copies the source operand to the destination operand. The add operator adds the source operand to the destination operand, while the sub operator subtracts the source from the destination. The cmp operator compares operands and sets condition codes. For example, if the left operand is greater than the right operand the condition codes ne, gt and ge will be set and the eq, lt, le condition codes will be unset. The true and false codes will be unaffected. The cmp instruction is used in conjunction with the *jp instruction. The final operator is gad, which stands for* Get ADdress. An instruction of this type loads the address of the right operand into the left operand which must be of type addr.

Note that in the mov, add and sub instructions there is no requirement that both operands be of the same type. Thus characters can be added to integers in checksum calculations.

7.4.3. ARITHMETIC ONE OPERAND INSTRUCTIONS

An arithmetic one operand instruction consists of an operator followed by a single operand. The operand has the same format as that used in two operand instructions. There are four operators : inc, dec, clr and arg. The inc operator increments its operand and the dec operator decrements it. The clr operator zero fills its operand. The arg operator places its operand onto the stack in preparation for a subroutine call.

7.4.4. CONTROL INSTRUCTIONS

There are many different control operators. Some are segment delimiters or markers. This type of operator has no operands. The beg operator marks the beginning of a subroutine. The data operator introduces a section of allocation instructions, while the text operator introduces a section of program code. These instructions are necessary since some machines, notably the VAX, require that programs and the variables they access be in different segments of memory. The endf marker denotes the end of the source file.

Labels are placed in a column to the left of the program code and are followed by a colon. When they are translated into a tuple they take the form

lab 0 0 <identifier> 0 .

There are three instructions which use labels: call, callc and jp. The call instruction consists of the operator followed by the label and an integer value. A call instruction branches to a subroutine and saves the return address on the stack. The instruction for returning from a subroutine is called ret and has no operands. The integer value in the call instruction contains the number of arguments. This was included because the calls instruction in VAX assembler requires this information as one of its operands. The callc instruction has the same format as the call instruction, but it is used to call logical functions which set the conditions codes true and false. The instruction for returning from a logical function is called retc. A retc instruction has one operand, true or false, depending on the required return value. If true is specified the true condition code is set and the false condition code is unset, and if false is specified false is set and true is unset.

The jump instruction, jp, consists of the operator followed by an optional qualifier followed by a label. If the qualifier is omitted this is an unconditional jump. However, if the qualifier is present it will be the name of one of the condition codes defined earlier which will have been set by a cmp instruction.

7.4.5. AN EXAMPLE

To conclude this discussion of the abstract machine an example program now follows. It adds a constant and a variable called avar to an integer whose address is 2 bytes into a character array.

```
        data
        array char 100 store
        variable int avar 3
        text
prog:   beg
        gad inx,store
        add 2[inx],(int)1
        add 2[inx],avar
        ret
```

This program can be expanded into the following tuples.

data	0	0	0	0
array	char	100	store	0
variable	int	0	avar	3
text	0	0	0	0
lab	0	0	prog	0
beg	0	0	0	0
gad	addr	inx	0	0
0	char	variable	store	0
add	int	inx_offset	0	2
0	int	const	0	1
add	int	inx_offset	0	2
0	int	variable	avar	0
ret	0	0	0	0

7.5. PRODUCTION OF I-CODE

The first stage in the translation process for PSL/2 is the production of I-code from the PSL/2 specification. This translation process can itself be divided into a number of steps.

The first step consists of producing an internal representation of the PSL/2 specification to serve as a database for I-code production. This internal representation is made up of a set of linked lists and trees. Firstly, there is a linked list containing an element for each frame type. Each element will contain a pointer to a linked list of field definitions and a pointer to a code tree of the receive action for this frame. There are also trees of events such as CLOSE_REQUEST and TIMER_EXPIRED. Finally, there is a symbol table which is a linked list of elements containing the following fields:

a) variable name

b) variable type - STATE, FLAG or SEQ_FIELD.

c) short name - for use in the target program.

The short name in each element is generated by the system

Once again LEX and YACC were used to generate a lexical analyser and a parser for PSL/2. The actions within the YACC specification contain code to build the structures described above.

Once the database has been constructed an algorithm is required to produce I-code from this information. As has been mentioned previously the FAPL compiler uses a set of macros to direct translation into PL/1. A similar approach was adopted for PSL/2. A collection of macros, collectively called a program template were written to translate the constructs of PSL/2 into the constructs of the abstract machine.

A program template consists of a list of keywords written in upper case identifying PSL/2 concepts together with templates of the I-code equivalent. In addition, comment lines may be included beginning with a single upper case letter C. An example template can be found in figure 7.4. In this example, a series of three dots indicates that text has been omitted. The order in which the sections are given is important and should be as described in the example. This is necessary as processing of later sections depends upon information contained in earlier sections.

The first section is where symbolic constants, such as buffer sizes, can be defined. The backslash at the end of a line supresses the trailing newline character, which would normally be part of the template. Following this the templates for the primitive actions of PSL/2 are defined. These templates may include calls to subroutines which are defined elsewhere.

After this templates are given for the comparison operators supported in PSL/2. These operators are equals (=) and not equals (<>). The template for the equals comparison is introduced by the token EQ, and the template for the not equals comparison is introduced by the token NE. Following this the templates for the logical operators OR, AND and NOT and the logical functions S_WINDOW_FULL and IN_R_WINDOW are given.

The section following this contains two templates for each type of field found in frame definitions. The SEND_CHARACTER and RECEIVING_CHARACTER templates are concerned with character constants. The following template describes the actions necessary to assemble a character constant into a frame for output. The template labeled RECEIVING_CHARACTER describes the actions required when a particular character is expected in a frame. The other

```
C
C       1) DEFINED CONSTANTS
C       --------------------
C
BUFN
3000\
BUFZ
300\
C
C       2) PRIMITIVE ACTIONS
C       --------------------
C
OPEN_R_WINDOW
        call openrw 0
NEW_STATE
        call stclr 0
        mov <state name>,1
SET
        mov <flag name>,1
UNSET
        mov <flag name>,0
INC
        inc <sequence field>
ENABLE_ABOVE
        call enable 0   /* user defined primative */
DISABLE_ABOVE
        call disble 0   /* user defined primative */
SEND_ABOVE
        call sendab 0
SEND_BELOW
        call sdb<frame name> 0
CANCEL
        arg <first parameter name>
        arg <second parameter name>
        call cancel 2
RETRAN
        arg <first parameter name>
        arg <second parameter name>
        call retran 2
RECEIVE
        call rec<frame name> 0
DISCARD
        call discrd 0
START_TIMER
        call statim 0   /* user defined primative */
STOP_TIMER
        call stptim 0   /* user defined primative */
ERROR
HALT
C
C       3a) COMPARISONS
C       ---------------
C
EQ
        mov <old stack top>,0
        cmp <left>,<right>
        jp ne <label>
        mov <old stack top>,1
<label>:\
NE
        mov <old stack top>,0
        cmp <left>,<right>
        jp eq <label>
        mov <old stack top>,1
<label>:\
C
C       3b) LOGICAL OPERATIONS
C       ----------------------
C
OR
        mov <old stack top>,1
        cmp <left>,1
        jp eq <label>
        cmp <right>,1
        jp eq <label>
```

FIGURE 7.4 - A PROGRAM TEMPLATE

```
        mov <old stack top>,0
<label>:\
AND
        mov <old stack top>,0
        cmp <left>,1
        jp ne <label>
        cmp <right>,1
        jp ne <label>
        mov <old stack top>,1
<label>:\
NOT
        mov <old stack top>,1
        cmp <operand>,1
        jp ne <label>
        mov <old stack top>,0
<label>:\
C
C       3c) LOGICAL FUNCTIONS
C       --------------------
C
S_WINDOW_FULL
        mov <old stack top>,0
        callc swfull 0
        jp false <label>
        mov <old stack top>,1
<label>:\
IN_R_WINDOW
        mov <old stack top>,0
        callc inrwnd 0
        jp false <label>
        mov <old stack top>,1
<label>:\
C
C       4) FRAME FIELDS
C       ---------------
C
SENDING_CHARACTER
        mov [inx],(char)<character>
        inc maxsm
        add calcks,[inx]
        inc inx
SENDING_VARIABLE
        mov [inx],<variable,name>
        add (char)[inx],64      /* character stuffing */
        inc maxsm
        add calcks,[inx]
        inc inx
SENDING_DATA
        call sdata 0
SENDING_PARAMS
        call sndpar 0
SENDING_CHECK_SUM
        call scks 0
RECEIVING_CHARACTER
        arg (int)1
        arg inx
        call readb 2
        add calcks,[inx]
        inc inx
RECEIVING_VARIABLE
        arg (int)1
        arg inx
        call readb 2
        add calcks,[inx]
        sub (char)[inx],64
        mov <variable name>,[inx]
RECEIVING_DATA
        call rdata 0
RECEIVING_PARAMS
        call getpar 0
RECEIVING_CHECK_SUM
        callc rcks 0
C
C       5) DECLARATIONS
C       ---------------
C
```

FIGURE 7.4 - A PROGRAM TEMPLATE (Cont.)

```
SEQ_FIELD
        variable seq <low-level name>   /* <high-level name> */
FLAG
        variable char <low-level name>  /* <high-level name> */
STATE
        variable char <low-level name>  /* <high-level name> */
C
C       6) STANDARD DECLARATIONS
C       ------------------------
C
STANDARD_DCLS
        array char BUFN swind   /* send window buffer variable */
        array char BUFN rwind   /* receive window buffer variable */
        array char BUFZ inpbuf  /* input buffer */
        array char BUFZ outbuf  /* output buffer */
        variable addr bsw -1    /* beginning of send window */
        variable addr tsw -1    /* top of send window */
        ...
C
C       7) OVERALL PROGRAM
C       ------------------
C
POLLING
tmain:  beg
        mbeg
        gad swinde,swind
        add swinde,BUFN
        gad rwinde,rwind
        add rwinde,BUFN
        call enable 0
m01:    jp m01

open:
        {ON_OPEN_REQUEST}
loop:   callc onca 0
        jp false l1
        {ON_CHARACTER_ABOVE}
l1:     callc oncb 0
        jp false l2
        {ON_CHARACTER_BELOW}
l2:     callc ctime 0
        jp false loop
        {ON_TIMER_EXPIRED}
        jp loop

close:  {ON_CLOSE_REQUEST}
        ret

EVENT_DRIVEN

tmain:  mbeg
        stim
        gad swinde,swind
        add swinde,BUFN
        gad rwinde,rwind
        add rwinde,BUFN
        call enable 0
        {ON_OPEN_REQUEST}
m01:    jp m01

close:  {ON_CLOSE_REQUEST}
        ret
chara:  {ON_CHARACTER_ABOVE}
        ret
charb:  {ON_CHARACTER_BELOW}
        ret
time:   {ON_TIMER_EXPIRED}
        ret

USER_ROUTINES
C
C       8) USER DEFINED ROUTINES
C       ------------------------
C

/*****  sndpar - send parameters *****/
```

FIGURE 7.4 - A PROGRAM TEMPLATE (Cont.)

```
sndpar: beg
        arg maxrw               /* put max window into buffer */
        arg inx
        call pi2 2
        add calcks,[inx]
        inc inx
        add calcks,[inx]
        inc inx
        arg maxrm               /* put max message into buffer */
        arg inx
        call pi2 2
        add calcks,[inx]
        inc inx
        add calcks,[inx]
        inc inx
        mov [inx],tindr         /* put into buffer terminator indication */
        add calcks,[inx]
        inc inx
        mov [inx],termr         /* put terminator into buffer */
        add (char)[inx],64      /* stuff terminator */
        add calcks,[inx]
        inc inx
        ret
        ...
```

FIGURE 7.4 - A PROGRAM TEMPLATE (Cont.)

templates in this section have similar functions.

After this can be found the formats of declarations for each PSL/2 variable type. Following this space is provided for standard variable declarations used in the code for the protocol entity.

Penultimately, there is a section concerned with the overall control structure of the program. Different target environments are supported by providing a choice of templates. Two types of possible environment were identified. These were the polling environment and the event-driven environment. In the polling environment the entity is being used within a substantial operating system which controls input and output buffering and allows polling of input queues. Supervisor or subroutine calls are used to interface with the operating system. The event-driven environment can be used where the host operating system provides a suitable interface or where the entity will be part of a device-driver or it is to be run on a computer without an operating system.

The final section contains service routines which are sufficiently machine-independent to be expressed in I-code. These routines are referenced in the previous sections.

The format of the templates themselves is fairly straightforward. They are sections of I-code in which various substitution strings have been placed. These substitutions strings are simply comments describing the substitution enclosed by braces, < and >. The content of the comment is unimportant since substitutions are made in a set order. In addition PSL/2 event names such as ON_CLOSE_REQUEST may be enclosed in curly brackets, { and }, and inserted into the template for the overall control structure. At these points within the template, code will be generated from the code tree for the actions associated with these events.

In the following stage of the translation process, a file is produced which combines information from the PSL/2 specification and the program template. This file consists of macro definitions, i-code instructions and macro calls. This file is later presented to the m4 macro-processor (Kernighan,1978). which

produces an i-code program. A systems flowchart summarising this activity can be found in figure 7.5.

In the stage one program, the constants, primative actions, comparisons, logical operations and functions, and declarations from the program template are translated into macros. Following this the code trees for the actions associated with events from the PSL/2 specification are converted into macros. This second set of macros contains calls to the first set. The remaining templates are processed and appended to these macros. The event names surrounded by curly brackets are converted into macro calls to the macros produced from the code trees. Finally, routines for sending and receiving each frame are generated. These routines use the macros generated from section 4 of the template. An example of an I-code program containing m4 macros is contained in figure 7.6. An example of an I-code program with the macros expanded is contained in figure 7.7.

7.6. TARGET ASSEMBLER SPECIFICATION

Given an abstract machine code program for a protocol entity, it is still necessary to translate it into the assembly code for a particular machine. As has been previously discussed in section 6.4, there is a need for a method of specifying this translation process. For PSL/2 a system was devised based upon pattern matching. A series of tables are supplied by the user which contain templates for target assembler translations of abstract machine instructions. During the translation I-code is transformed into tuples and each tuple or tuple pair is processed by reference to these tables. This is achieved, firstly, by using the operator to access the appropriate table and, secondly, by using the type and mode elements of the tuple to search for the correct template within that table.

The target assembler specification itself consists of two sections. In the first section composite modes and types can be defined. These are groups of modes and types connected by the OR symbol |. This is similar to a macro facility, since it enables the specifier to define names that will be expanded during the translation process.

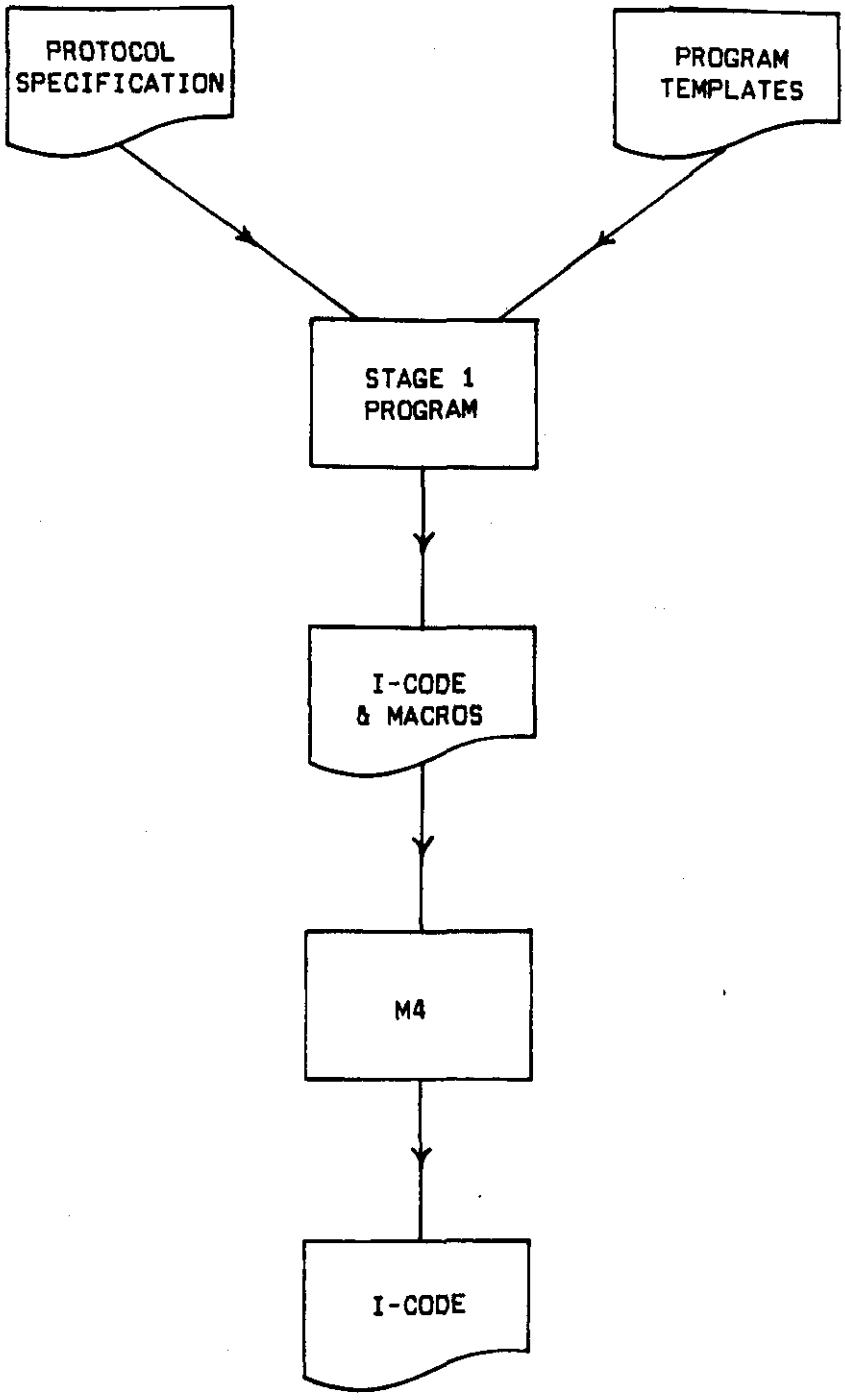


FIGURE 7.5 - PSL/2 TRANSLATION - LDT

```
        undefine(`index')
        undefine(`len')
        undefine(`eval')
define(BUFN,`3000')             --[ Beginning of first set of macros
define(BUFZ,`300')
define(OCC,`0')
define(SEQU,`1')
define(TIMER,`2')
define(LEN,`6')
define(FRAME,`10')
define(OPEN_R_WINDOW,
                        /* `$0' */
        call openrw 0
')
define(NEW_STATE,
                        /* `$0' */
        call stclr 0
        mov $1,1
')
define(EQ,
                        /* `$0' */
        mov $1,0
        cmp $2,$3
        jp ne $4
        mov $5,1
$6:')
define(SENDING_CHARACTER,
                        /* `$0' */
        mov [inx],(char)$1
        inc maxsm
        add calcks,[inx]
        inc inx
')
define(SENDING_VARIABLE,
                        /* `$0' */
        mov [inx],$1
        add (char)[inx],64      /* character stuffing */
        inc maxsm
        add calcks,[inx]
        inc inx
')
define(SENDING_DATA,
                        /* `$0' */
        call sdata 0
')
define(SEQ_FIELD,`      variable seq $1 /* $2 */
')
define(FLAG,`   variable char $1        /* $2 */
')
define(STATE,`  variable char $1        /* $2 */
')
define(ON_OPEN_REQUEST,         --[ Beginning of second set of macros
                        /* `$0' */
OPEN_R_WINDOW
DISABLE_ABOVE
SEND_BELOW(02)
START_TIMER
NEW_STATE(st00)
')
define(ON_CHARACTER_BELOW,
                        /* `$0' */
        gad inx,inpbuf
        arg inx
        call reada 1
        cmp (char)[inx],2
        jp ne ocb00
if03:   mov tv00,st01
        cmp tv00,1
        jp ne else03
then03:
RECEIVE(00)
if04:IN_R_WINDOW(tv00,lab05,tv00,lab05)
        cmp tv00,1
        jp ne fi04
then04:
SEND_ABOVE
```

FIGURE 7.6 - I-CODE PROGRAM WITH MACROS

```
INC(seq03)
fi04:
SEND_BELOW(01)
else03:
DISCARD
fi03:
ocb00:  cmp (char)[inx],6
        jp ne ocb01
if06:   mov tv00,st01
        cmp tv00,1
        jp ne else06
then06:
RECEIVE(01)
if07:NE(tv00,seq02,seq04,lab08,tv00,lab08)
        cmp tv00,1
        jp ne fi07
then07:
CANCEL(seq02,seq02)
INC(seq04)
ENABLE_ABOVE
fi07:
else06:
DISCARD
fi06:
ocb04:  cmp (char)[inx],5
        jp ne ocb05
if17:   mov tv00,st02
        cmp tv00,1
        jp ne else17
then17:
HALT
else17:
DISCARD
fi17:
ocb05:')
        data
        array char BUFN swind    /* send window buffer variable */
        array char BUFN rwind    /* receive window buffer variable */
        array char BUFZ inpbuf   /* input buffer */
        array char BUFZ outbuf   /* output buffer */
        variable addr bsw -1     /* beginning of send window */
        variable addr tsw -1     /* top of send window */
FLAG(tv00)
FLAG(tv01)
FLAG(tv02)
SEQ_FIELD(seq00,send_no)
SEQ_FIELD(seq01,recv_no)
SEQ_FIELD(seq02,ack_no)
SEQ_FIELD(seq03,exp_no)
SEQ_FIELD(seq04,ack_exp_no)
FLAG(fl00,i_know)
FLAG(fl01,i_am_known)
STATE(st00,opening)
STATE(st01,data_transfer)
STATE(st02,closing)
        text
tmain:  beg
        gad swinde,swind
        add swinde,BUFN
        gad rwinde,rwind
        add rwinde,BUFN
        call enable 0
m01:    jp m01

open:
        ON_OPEN_REQUEST
loop:   call onca 0
        jp false l1
        ON_CHARACTER_ABOVE
l1:     call oncb 0
        jp false l2
        ON_CHARACTER_BELOW
l2:     call ctime 0
        jp false loop
        ON_TIMER_EXPIRED
        jp loop
```

FIGURE 7.6 - I-CODE PROGRAM WITH MACROS (Cont.)

```
close:  ON_CLOSE_REQUEST
        ret

/*****  sndpar - send parameters *****/

sndpar: beg
        arg maxrw               /* put max window into buffer */
        arg inx
        call pi2 2
        add calcks,[inx]
        inc inx
        add calcks,[inx]
        inc inx
        arg maxrm               /* put max message into buffer */
        arg inx
        call pi2 2
        add calcks,[inx]
        inc inx
        add calcks,[inx]
        inc inx
        mov [inx],tindr         /* put into buffer terminator indication */
        add calcks,[inx]
        inc inx
        mov [inx],termr         /* put terminator into buffer */
        add (char)[inx],64      /* stuff terminator */
        add calcks,[inx]
        inc inx
        ret
        ...

/***** clear state variables *****/

stclr:  beg
        clr st00
        clr st01
        clr st02
        ret

/***** receive info *****/

rec00:  beg
        gad inx,inpbuf
        mov seqno,[inx]
RECEIVING_VARIABLE(seq01)
RECEIVING_DATA
RECEIVING_CHECK_SUM
        ret


/***** receive ack *****/

rec01:  beg
        gad inx,inpbuf
RECEIVING_VARIABLE(seq02)
RECEIVING_CHECK_SUM
        ret
        ...

/***** discrd - routine to skip until frame id *****/

discrd: beg
dis01:  gad tema,ch
        arg tema
        call rcb 1
        jp dis01
dis02:  ret

/***** sending info *****/

sdb00:  beg
        add tsw,BUFZ
        cmp tsw,swinde
        jp ne s00
        gad tsw,swind
s00:    inc actsw
        mov inx,tsw
```

FIGURE 7.6 - I-CODE PROGRAM WITH MACROS (Cont.)

```
        mov (char)OCC[inx],1
        mov SEQU[inx],seq00
        mov TIMER[inx],timint
        add inx,FRAME
        mov begf,inx
SENDING_CHARACTER(2)
SENDING_VARIABLE(seq00)
SENDING_DATA
SENDING_CHECK_SUM
        mov lenf,inx
        sub inx,begf
        mov inx,tsw
        mov LEN[inx],lenf
        arg begf
        arg lenf
        call rddb 2
        ret

/***** sending ack *****/

sdb01:  beg
        gad inx,outbuf
        mov begf,inx
SENDING_CHARACTER(6)
SENDING_VARIABLE(seq01)
SENDING_CHECK_SUM
        mov lenf,inx
        sub inx,begf
        arg begf
        arg lenf
        call rddb 2
        ret
        ...
```

FIGURE 7.6 - I-CODE PROGRAM WITH MACROS (Cont.)

```
        data
        array char 3000 swind   /* send window buffer variable */
        array char 3000 rwind   /* receive window buffer variable */
        array char 300 inpbuf   /* input buffer */
        array char 300 outbuf   /* output buffer */
        variable addr bsw -1    /* beginning of send window */
        variable addr tsw -1    /* top of send window */
        ...

        variable char tv00      /*  */

        variable char tv01      /*  */

        variable char tv02      /*  */

        variable seq seq00      /* send_no */

        variable seq seq01      /* recv_no */

        variable seq seq02      /* ack_no */

        variable seq seq03      /* exp_no */

        variable seq seq04      /* ack_exp_no */

        variable char fl00      /* i_know */

        variable char fl01      /* i_am_known */

        variable char st00      /* opening */

        variable char st01      /* data_transfer */

        variable char st02      /* closing */

        text
tmain:  beg
        mbeg
        gad swinde,swind
        add swinde,3000
        gad rwinde,rwind
        add rwinde,3000
        call enable 0
m01:    jp m01

open:
                                /* ON_OPEN_REQUEST */
                        /* OPEN_R_WINDOW */
        call openrw 0

                        /* DISABLE_ABOVE */
        call disble 0   /* user defined primative */

                        /* SEND_BELOW */
        call sdb02 0

                        /* START_TIMER */
        call statim 0   /* user defined primative */

                        /* NEW_STATE */
        call stclr 0
        mov st00,1


loop:   callc onca 0
        jp false l1
                                /* ON_CHARACTER_ABOVE */
if00:   mov tv00,st01
        cmp tv00,1
        jp ne else00
then00:
                        /* SEND_BELOW */
        call sdb00 0

if01:                   /* S_WINDOW_FULL */
        mov tv00,0
```

FIGURE 7.7 - AN I-CODE PROGRAM WITH MACROS EXPANDED

```
        callc swfull 0
        jp false lab02
        mov tv00,1
lab02:
        cmp tv00,1
        jp ne fi01
then01:
                        /* DISABLE_ABOVE */
        call disble 0   /* user defined primative */

fi01:
else00:
                        /* ERROR */

fi00:

l1:     callc oncb 0
        jp false l2
                                /* ON_CHARACTER_BELOW */
        gad inx,inpbuf
        arg inx
        call reada 1
        cmp (char)[inx],2
        jp ne ocb00
if03:   mov tv00,st01
        cmp tv00,1
        jp ne else03
then03:
                        /* RECEIVE */
        call rec00 0

if04:                   /* IN_R_WINDOW */
        mov tv00,0
        callc inrwnd 0
        jp false lab05
        mov tv00,1
lab05:
        cmp tv00,1
        jp ne fi04
then04:
                        /* SEND_ABOVE */
        call sendab 0

                        /* INC */
        inc seq03

fi04:
                        /* SEND_BELOW */
        call sdb01 0

else03:
                        /* DISCARD */
        call discrd 0

fi03:
ocb00:  cmp (char)[inx],6
        jp ne ocb01
if06:   mov tv00,st01
        cmp tv00,1
        jp ne else06
then06:
                        /* RECEIVE */
        call rec01 0

if07:                   /* NE */
        mov tv00,0
        cmp seq02,seq04
        jp eq lab08
        mov tv00,1
lab08:
        cmp tv00,1
        jp ne fi07
then07:
                        /* CANCEL */
        arg seq02
        arg seq02
```

FIGURE 7.7 - AN I-CODE PROGRAM WITH MACROS EXPANDED (cont.)

```
        call cancel 2

                        /* INC */
        inc seq04

                        /* ENABLE_ABOVE */
        call enable 0   /* user defined primative */

fi07:
else06:
                        /* DISCARD */
        call discrd 0

fi06:
ocb01:  cmp (char)[inx],1
        jp ne ocb02
if09:   mov tv00,st00
        cmp tv00,1
        jp ne else09
then09:
                        /* RECEIVE */
        call rec02 0

                        /* SEND_BELOW */
        call sdb03 0

                        /* START_TIMER */
        call statim 0   /* user defined primative */

                        /* SET */
        mov fl00,1

if10:   mov tv01,fl01
        mov tv02,fl00
                        /* AND */
        mov tv00,0
        cmp tv02,1
        jp ne labl1
        cmp tv01,1
        jp ne labl1
        mov tv00,1
labl1:
        cmp tv00,1
        jp ne fi10
then10:
                        /* NEW_STATE */
        call stclr 0
        mov st01,1

                        /* ENABLE_ABOVE */
        call enable 0   /* user defined primative */

                        /* STOP_TIMER */
        call stptim 0   /* user defined primative */

fi10:
else09:
                        /* DISCARD */
        call discrd 0

fi09:
ocb02:  cmp (char)[inx],3
        jp ne ocb03
if12:   mov tv00,st00
        cmp tv00,1
        jp ne else12
then12:
                        /* RECEIVE */
        call rec03 0

                        /* SET */
        mov fl01,1

if13:   mov tv01,fl01
        mov tv02,fl00
                        /* AND */
```

FIGURE 7.7 - AN I-CODE PROGRAM WITH MACROS EXPANDED (cont.)

```
        mov tv00,0
        cmp tv02,1
        jp ne lab14
        cmp tv01,1
        jp ne lab14
        mov tv00,1
lab14:
        cmp tv00,1
        jp ne fil3
then13:
                        /* NEW_STATE */
        call stclr 0
        mov st01,1

                        /* ENABLE_ABOVE */
        call enable 0   /* user defined primative */

                        /* STOP_TIMER */
        call stptim 0   /* user defined primative */

fil3:
else12:
                        /* DISCARD */
        call discrd 0

fil2:
ocb03:  cmp (char)[inx],4
        jp ne ocb04
                        /* RECEIVE */
        call rec04 0

if15:                   /* NE */
        mov tv00,0
        cmp seq01,seq03
        jp eq lab16
        mov tv00,1
lab16:
        cmp tv00,1
        jp ne fil5
then15:
                        /* SEND_BELOW */
        call sdb05 0

                        /* HALT */

fil5:
ocb04:  cmp (char)[inx],5
        jp ne ocb05
if17:   mov tv00,st02
        cmp tv00,1
        jp ne else17
then17:
                        /* HALT */

else17:
                        /* DISCARD */
        call discrd 0

fil7:
ocb05:
12:     callc ctime 0
        jp false loop
                                /* ON_TIMER_EXPIRED */
if18:   mov tv00,st00
        cmp tv00,1
        jp ne fil8
then18:
                        /* SEND_BELOW */
        call sdb02 0

                        /* START_TIMER */
        call statim 0   /* user defined primative */

fil8:

        jp loop
```

FIGURE 7.7 - AN I-CODE PROGRAM WITH MACROS EXPANDED (cont.)

```
close:                          /* ON_CLOSE_REQUEST */
                        /* SEND_BELOW */
        call sdb04 0

                        /* NEW_STATE */
        call stclr 0
        mov st02,1


        ret

/*****  sndpar - send parameters *****/

sndpar: beg
        arg maxrw               /* put max window into buffer */
        arg inx
        call pi2 2
        add calcks,[inx]
        inc inx
        add calcks,[inx]
        inc inx
        arg maxrm               /* put max message into buffer */
        arg inx
        call pi2 2
        add calcks,[inx]
        inc inx
        add calcks,[inx]
        inc inx
        mov [inx],tindr         /* put into buffer terminator indication */
        add calcks,[inx]
        inc inx
        mov [inx],termr         /* put terminator into buffer */
        add (char)[inx],64      /* stuff terminator */
        add calcks,[inx]
        inc inx
        ret

/***** clear state variables *****/

stclr:  beg
        clr st00
        clr st01
        clr st02
        ret

/***** receive info *****/

rec00:  beg
        gad inx,inpbuf
        mov seqno,[inx]
                        /* RECEIVING_VARIABLE */
        arg (int)1
        arg inx
        call readb 2
        add calcks,[inx]
        sub (char)[inx],64
        mov seq01,[inx]

                        /* RECEIVING_DATA */
        call rdata 0

                        /* RECEIVING_CHECK_SUM */
        callc rcks 0

        ret


/***** receive ack *****/

rec01:  beg
        gad inx,inpbuf
                        /* RECEIVING_VARIABLE */
        arg (int)1
        arg inx
        call readb 2
```

FIGURE 7.7 - AN I-CODE PROGRAM WITH MACROS EXPANDED (cont.)

```
        add calcks,[inx]
        sub (char)[inx],64
        mov seq02,[inx]

                        /* RECEIVING_CHECK_SUM */
        callc rcks 0

        ret
        ...

/***** discrd - routine to skip until frame id *****/

discrd: beg
dis01:  gad tema,ch
        arg tema
        call rcb 1
        jp dis01
dis02:  ret

/***** sending info *****/

sdb00:  beg
        add tsw,300
        cmp tsw,swinde
        jp ne s00
        gad tsw,swind
s00:    inc actsw
        mov inx,tsw
        mov (char)0[inx],1
        mov 1[inx],seq00
        mov 2[inx],timint
        add inx,10
        mov begf,inx
                        /* SENDING_CHARACTER */
        mov [inx],(char)2
        inc maxsm
        add calcks,[inx]
        inc inx

                        /* SENDING_VARIABLE */
        mov [inx],seq00
        add (char)[inx],64      /* character stuffing */
        inc maxsm
        add calcks,[inx]
        inc inx

                        /* SENDING_DATA */
        call sdata 0

                        /* SENDING_CHECK_SUM */
        call scks 0

        mov lenf,inx
        sub inx,begf
        mov inx,tsw
        mov 6[inx],lenf
        arg begf
        arg lenf
        call rddb 2
        ret

/***** sending ack *****/

sdb01:  beg
        gad inx,outbuf
        mov begf,inx
                        /* SENDING_CHARACTER */
        mov [inx],(char)6
        inc maxsm
        add calcks,[inx]
        inc inx

                        /* SENDING_VARIABLE */
        mov [inx],seq01
        add (char)[inx],64      /* character stuffing */
        inc maxsm
```

FIGURE 7.7 - AN I-CODE PROGRAM WITH MACROS EXPANDED (cont.)

```
add calcks,[inx]
inc inx

                    /* SENDING_CHECK_SUM */
call scks 0

mov lenf,inx
sub inx,begf
arg begf
arg lenf
call rddb 2
ret
...
```

FIGURE 7.7 - AN I-CODE PROGRAM WITH MACROS EXPANDED (cont.)

The second section contains the tables of template and pattern matching information. A template is a string of characters surrounded by double quotes. The string may contain two types of special character sequence. Firstly, there are notations for the unprintable characters tab and newline. These follow the C language conventions:

\t - tab character
\n - newline character

Secondly, there are substitutions. These are introduced by a question mark and are defined independently for each table. The exact format of the pattern matching information depends on the particular group of instructions covered by the table.

The table for allocation instructions is called ALLOC. In this table the matching information is simply the particular type or types to which this template can be applied. For example, if a character and a sequence number are both represented by a single byte on a particular machine and both addresses and integers are held in a single word, the user may have specified the composite types byte and word using the following definitions.

```
DEF_TYPE byte = CHAR|SEQ
DEF_TYPE word = ADDR|INT
```

The alloc table might be written as:

```
TABLE ALLOC
        ARRAY,CHAR,"?n BSS ?l\n"
        VARIABLE,byte,"?n BYTE ?v\n"
        VARIABLE,word,"?n DATA ?v\n"
END_TABLE
```

In this example, only character arrays are being supported, while simple variables of all types are supported. If the two composite types had not been defined the table would have had to be written as:

```
TABLE ALLOC
        ARRAY,CHAR,"?n BSS ?l\n"
        VARIABLE,CHAR|SEQ,"?n BYTE ?v\n"
        VARIABLE,ADDR|INT,"?n DATA ?v\n"
END_TABLE
```

This example illustrates the placing of substitution code within templates. There are three codes defined for allocation

instructions. The code ?n is the name of an identifier, while the code ?l is the length of an array and the code ?v is the initial value of an variable.

The table for arithmetic two operand operators is called ARITH_TWO_OP. Each line of this table requires a pair of type and mode definitions for each template. The first pair is concerned with the left operand, while the second pair is concerned with the right operand. The substitution codes are ?1 for the left operand and ?2 for the right operand. The actual form of the substitution will be deduced from a separate table called SUBS. The structure of a complete table would be

```
TABLE ARITH_TWO_OP
        MOV,byte,VAR|INX_USE|INX_OFF,
            byte,VAR|INX_USE|INX_OFF|PARAM,
            "MOVB ?2,?1\n"
        MOV,byte,VAR|INX_USE|INX_OFF,
            byte,CONST,
            "LI R0,?2
            .
            .
        GAD,ADDR,anymode,anytype,VAR,
            "LI R0,?2\n MOV R0,?1\n"
END_TABLE
```

The table for arithmetic one operand operators is called ARITH_ONE_OP. Each line contains a single set of type and mode definitions. The substitution code for the single operand is ? .

There is a table for JP instructions where each template is accessed by qualifier name. The word ANY is used to signify the template which is to be used for unconditional branch instructions. A JP table is given below:

```
TABLE JP
        ANY," JMP ?\n"
        EQ, " JEQ ?\n"
        NE, " JNE ?\n"
        LT, " JLT ?\n"
        GT, " JGT ?\n"
        LE, " JGE ?\n"
        GE, " JGE ?\n"
        TRUE, " JEQ ?\n"
        FALSE," JNE ?\n"
END_TABLE
```

The RETC table has the same format as the JP table, except that the only qualifiers allowed are TRUE and FALSE.

Some operators are not qualified by type and mode information or condition code qualifiers. For these operators single entry tables introduced by the word TEMPLATE are used. The operators which use this type of table are beg, call, callc, ret, text, data and endf. Of these operators only call and callc require substitution codes. The codes used are ?l denoting the position of the name of the subroutine and ?a denoting the number of arguments. Example templates for beg and call are given below.

```
TEMPLATE BEG  ".word 0x00\n"
TEMPLATE CALL "calls $?a,?l\n"
```

Templates are required for label declaration and label use. These templates are called LAB_DCL and LAB_USE respectively. The template for label declaration defines the format of label declarations in the target assembler, for example, whether they are followed by a colon or some other character. The substitution code ? yields the name of the label. The template for label use defines the format of labels when they appear as operands in branch instructions and uses the same substitution code as is used for label declaration.

```
TEMPLATE LAB_DCL "_?:\n"
TEMPLATE LAB_USE "_?"
```

The SUBS table mentioned earlier is similar to the LAB_USE template. It is indexed by the six access modes and caters for the differing formats of addressing modes within the target assembler. The table below gives details of substitution codes for each mode.

mode	substitution code	value
variable	?	identifier name
inx_use		no substitution
inx_offset	?	offset
inx		no substitution
param	?	parameter number
const	?	constant value

Two templates must be provided. The first template is used to indicate instructions that must precede the translation of the current instruction. This template may be used to set up indices. The second template indicates the format of the translation of the current instruction.

```
TABLE SUBS
        VAR,      "",".?"
        INX_USE,  "","*R6"
        INX_OFF,  "",".?(R6)"
        INX,      "","R6"
        PARAM,    " MOV R12,R0\n AI R0,-6-?-?\n MOV *R0,R0","R0"
        CONST,    "","?"
END_TABLE
```

An example of a target assembler specification for a Texas Instruments computer is given in figure 7.8.

7.7. I-CODE TO TARGET ASSEMBLER TRANSLATION

The second stage translation has to be repeated for each type of machine in the network. The first part of this stage consists of reading the target assembler specification and producing a database for I-code translation. YACC and LEX were once again used to generate a lexical analysis and a parser for the target assembler specification. Once this specification has been read and validated a set of internal tables should contain all that is necessary to produce a target assembler version of the I-code program.

YACC and LEX were used to generate a lexical analyser and a parser for the I-code language. The I-code program is converted into a five-element tuple form. For each tuple or tuple pair the appropriate tables are searched for a match on certain elements of the tuple. When the appropriate template is found it is written to the output file and the specified substitutions are made. A system flowchart for this stage of the translation process can be found in figure 7.9.

7.8. MACHINE-DEPENDENT INTERFACE ROUTINES

The user is required to provide a set of routines to act as an interface with the host operating system. The exact requirements will depend upon the contents of the program template. Routines for reading from and writing to the communications medium

```
DEF_TYPE byte=CHAR|SEQ

DEF_TYPE word=ADDR|INT

DEF_TYPE anytype=byte|word

DEF_MODE gen=VAR|INX_USE|INX_OFF

DEF_MODE anymode=gen|PARAM|INX

TABLE ALLOC
    ARRAY,CHAR,"?n BSS ?1\n"
    VARIABLE,byte,"?n BYTE ?v\n"
    VARIABLE,word,"?n DATA ?v\n"
END_TABLE

TABLE ARITH_TWO_OP
    MOV,byte,gen,byte,gen|PARAM,   " MOVB ?2,?1\n"
    MOV,byte,gen,byte,CONST,       " LI R0,?2\n SWPB R0 MOVB R0,?1\n"
    MOV,word,gen|INX,word,anymode," MOV ?2,?1\n"
    MOV,word,gen|INX,word,CONST,   " LI R0,?2\n MOV R0,?1\n"
    ADD,byte,gen,byte,gen|PARAM,   " AB ?2,?1\n"
    ADD,byte,gen,byte,CONST,       " LI R0,?2\n SWPB R0 AB R0,?1\n"
    ADD,word,gen|INX,word,anymode," A ?2,?1\n"
    ADD,word,gen|INX,word,CONST,   " LI R0,?2\n A R0,?1\n"
    ADD,INT,gen,CHAR,gen,          " MOVB ?2,R0\n SRL R0,8\n A R0,?1\n"
    ADD,INT,gen,CHAR,CONST,        " LI R0,?2\n A R0,?1\n"
    SUB,byte,gen,byte,gen|PARAM,   " SB ?2,?1\n"
    SUB,byte,gen,byte,CONST,       " LI R0,?2 SWPB R0\n SB R0,?1\n"
    SUB,word,gen|INX,word,anymode," S ?2,?1\n"
    SUB,word,gen|INX,word,CONST,   " LI R0,?2\n S R0,?1\n"
    CMP,byte,gen,byte,gen|PARAM,   " CB ?2,?1\n"
    CMP,byte,gen,byte,CONST,       " LI R0,?2\n SWPB R0\n CB R0,?1\n"
    CMP,word,gen|INX,word,anymode," C ?2,?1\n"
    CMP,word,gen|INX,word,CONST,   " LI R0,?2\n C R0,?1\n"
    GAD,ADDR,anymode,anytype,VAR ," LI R0,?2\n MOV R0,?1\n"
    GAD,ADDR,anymode,anytype,INX_USE ," MOV R6,?1\n"
END_TABLE

TABLE ARITH_ONE_OP
    INC,byte,gen," MOVB ?,R0\n SRL R0,8\n INC R0\n SWPB R0\n MOVB R0,?\n"
    INC,word,gen|INX," INC ?\n"
    DEC,byte,gen," MOVB ?,R0\n SRL R0,8\n DEC R0\n SWPB R0\n MOVB R0,?\n"
    DEC,word,gen|INX," INC ?\n"
    ARG,byte,anymode," CLR R0\n MOVB ?,R0\n MOV R0,*R10+\n"
    ARG,word,gen|INX," MOV ?,*R10+\n"
    ARG,word,CONST  ," LI R0,?\n MOV R0,*R10+\n"
    CLR,byte,gen    ," CLR R0\n MOVB R0,?"
    CLR,word,gen|INX," CLR ?\n"
END_TABLE

TABLE JP
    ANY," JMP ?\n"
    EQ ," JEQ ?\n"
    NE ," JNE ?\n"
    LT ," JLT ?\n"
    GT ," JGT ?\n"
    LE ," JLE ?\n"
    GE ," JGE ?\n"
    TRUE ," JEQ ?\n"
    FALSE," JNE ?\n"
END_TABLE

TABLE RETC
    TRUE ," LI R0,1\n RT\n"
    FALSE," CLR R0\n RT\n"
END_TABLE

TEMPLATE LAB_DCL "?\n"

TEMPLATE LAB_USE "?"

TEMPLATE BEG " MOV R11,*R10+\n MOV R10,R12\n"

TEMPLATE CALL " LI R0,?a+?a\n MOV R0,*R10+\n BL .?1\n"
```

FIGURE 7.8 - TEXAS TARGET ASSEMBLER SPECIFICATION

```
TEMPLATE CALLC " LI R0,?a+?a\n MOV R0,*R10+\n BL .?l\n CI R0,0\n"

TEMPLATE RET " DECT R10\n MOV *R10,R11\n DECT R10\n S *R10,R10\n RT\n"

TEMPLATE TEXT ""

TEMPLATE DATA ""

TEMPLATE ENDF ""

TABLE SUBS
    VAR     ,"",".?"
    INX_USE,"","*R6"
    INX_OFF,"",".?(R6)"
    INX     ,"","R6"
    PARAM   ," MOV R12,R0\n AI R0,-6-?-?\n MOV *R0,R0","R0"
    CONST   ,"","?"
END_TABLE
```

FIGURE 7.8 - TEXAS TARGET ASSEMBLER SPECIFICATION (cont.)

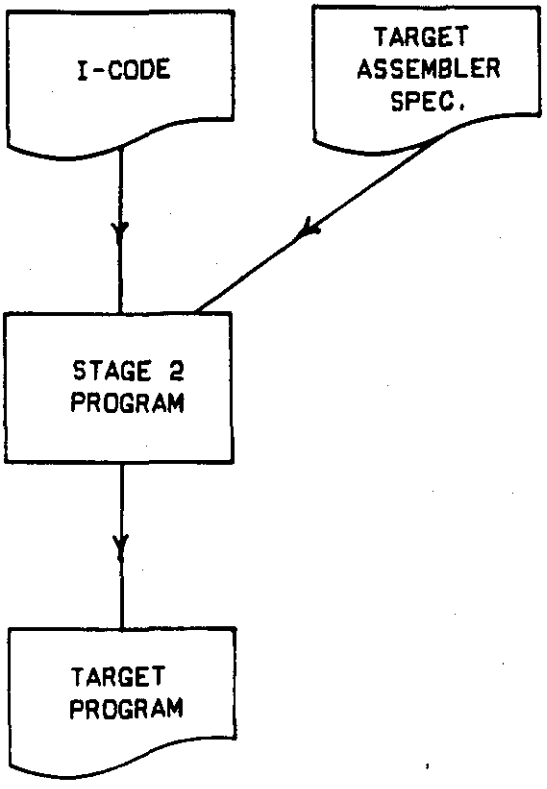


FIGURE 7.9 - PSL/2 TRANSLATION - MDT

and routines for communicating with the layer above would certainly be necessary. An example set of routines is given in figure 7.10. They are written in the C programming language for the UNIX operating system.

7.9. IMPLEMENTATION AND MAINTENANCE

The system as outlined here has a number of strengths. It has been devised in such a way that different people can use their own expertise in part of the design and implementation of the protocol, without needing to know about every aspect of the work. This is illustrated in figure 7.11. The protocol designer can produce a specification without knowing how PSL/2 is implemented in terms of I-code. Such implementation details are tackled by the protocol implementation designer. The implementation designer will also produce a specification of the routines to interface with the host operating system. An expert in the assembler of a particular machine can implement these routines and produce a target assembler specification. Thus protocol implementation is split into several discrete tasks, according to the old maxim "divide and conquer".

The initial implementation of a network can proceed as follows:

(1) A protocol is designed and specified.

(2) A program template is written.

(3) A target assembler specification is written for each machine together with a set of operating system interface routines.

(4) The protocol specification and program template are submitted to the first stage of the retargetable compiler and an i-code program is produced.

(5) Each target assembler specification is submitted to the second stage of the compiler together with the i-code program produced in the first stage. This produces an assembler equivalent of the i-code program for each machine in the network.

```
£include <stdio.h>
£include <sgtty.h>

typedef union {
                struct {
                        unsigned p3 :6;
                        unsigned p2 :6;
                        unsigned p1 :6;
                        unsigned    :14;
                } div;
                int word;
} WORD_DIV;

int above;      /* switch for above */

/****** LISTEN TO ABOVE ******/

int onca(){
        int temp;
        if(ioctl(0,FIONREAD,&temp)==-1)return(0);
        if(temp>0) return(1);
        else return(0);
}

/****** LISTEN TO BELOW ******/

int oncb(count)
int count;
{
        int temp;
        if(ioctl(3,FIONREAD,&temp)==-1)return(0);
        if(temp>=count) return(1);
        else return(0);
}

/****** READ CHARACTER FROM ABOVE ******/

rca(dest)
char *dest;
{
        read(0,dest,1);
}

/****** SEND CHARACTER TO ABOVE ******/

sca(dest)
char *dest;
{
        write(1,dest,1);
}

/****** READ FROM BELOW ******/

readb(dest,count)
char *dest;
int count;
{
        read(3,dest,count);
}

/****** SEND TO BELOW ******/

sddb(source,count)
char *source;
int count;
{
        write(5,source,count);
        write(4,source,count);
}

/****** ENABLE ABOVE ******/

enable(){
        above=1;
}
```

FIGURE 7.10 - UNIX SYSTEMS INTERFACE

```
/****** DISABLE ABOVE ******/

disable(){
        above=0;
}

/****** GET ONE BYTE STUFFED INTEGER ******/

gi1(buf,dest)
char *buf;
WORD_DIV *dest;
{
        dest->word=0;
        rddb(buf,1);
        dest->div.p3=(int)(*buf&077);
}

/****** GET TWO BYTE STUFFED INTEGER *******/

gi2(buf,dest)
char *buf;
WORD_DIV *dest;
{
        dest->word=0;
        rddb(buf,1);
        dest->div.p2=(int)(*buf++&077);
        rddb(buf,1);
        dest->div.p3=(int)(*buf&077);
}

/****** GET THREE BYTE STUFFED INTEGER ******/

gi3(buf,dest)
char *buf;
WORD_DIV *dest;
{
        dest->word=0;
        rddb(buf,1);
        dest->div.p1=(int)(*buf++&077);
        rddb(buf,1);
        dest->div.p2=(int)(*buf++&077);
        rddb(buf,1);
        dest->div.p3=(int)(*buf&077);
}

/****** PUT ONE BYTE STUFFED INTEGER ******/

pi1(buf,source)
char *buf;
WORD_DIV source;
{
        *buf++=((char)(source.div.p3))|0100;
}

/****** PUT TWO BYTE STUFFED INTEGER ******/

pi2(buf,source)
char *buf;
WORD_DIV source;
{
        *buf++=((char)(source.div.p2))|0100;
        *buf++=((char)(source.div.p3))|0100;
}

/****** PUT THREE BYTE STUFFED INTEGER ******/

pi3(buf,source)
char *buf;
WORD_DIV source;
{
        *buf++=((char)(source.div.p1))|0100;
        *buf++=((char)(source.div.p2))|0100;
        *buf++=((char)(source.div.p3))|0100;
}
```

FIGURE 7.10 - UNIX SYSTEMS INTERFACE (Cont.)

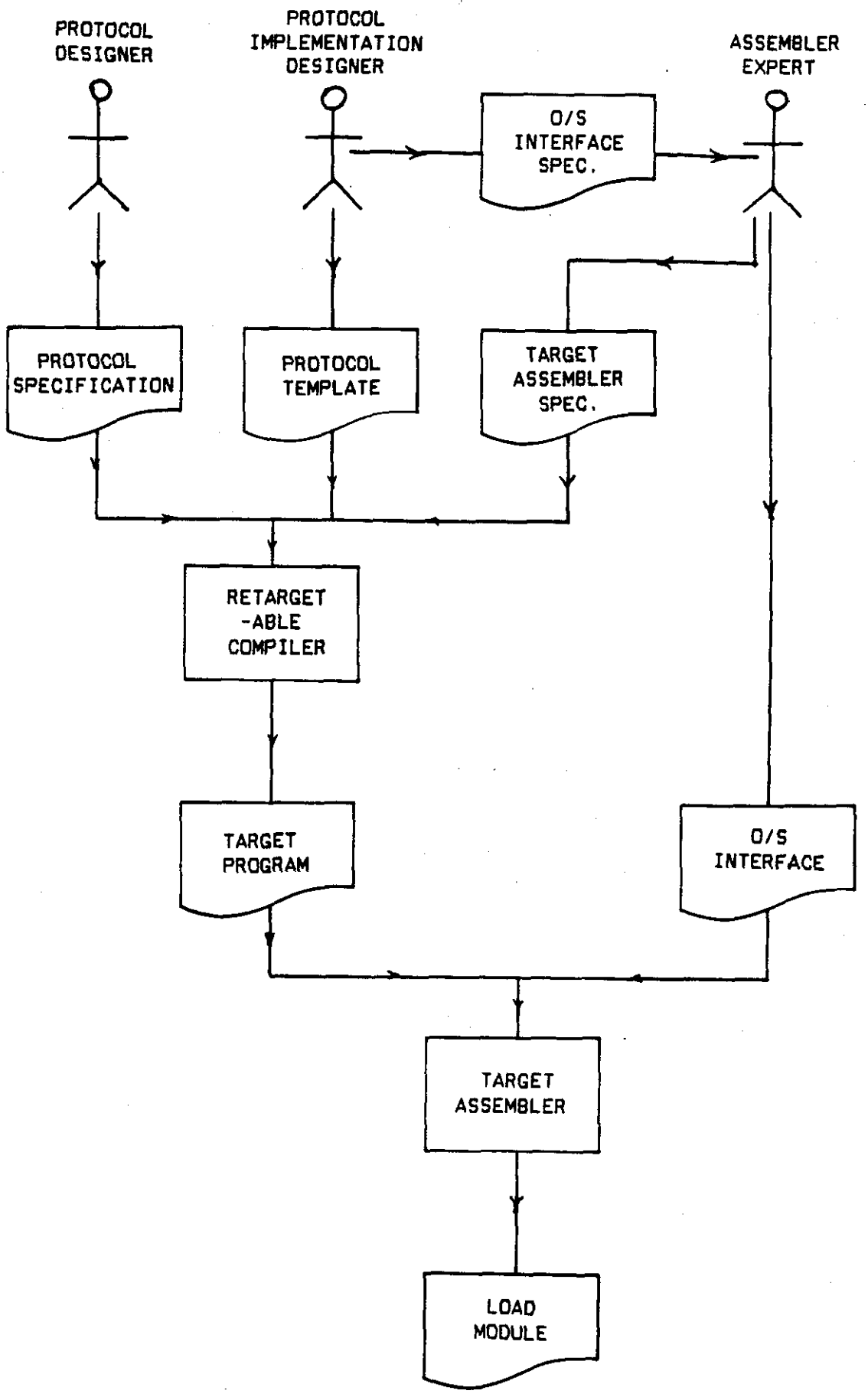


FIGURE 7.11 - PROTOCOL IMPLEMENTATION USING PSL/2

(6) The assembler programs are transferred onto their target machines. This may involve using magnetic tape, floppy disks or communication lines.

(7) The operating system interface routines are input to their target machines.

(8) The assembler programs and interface routines on each machine are assembled and linked producing a protocol program for each machine.

The modularity of this system makes maintenance straightforward. The addition of a new machine to the network requires that a new target assembler specification is written, together with a set of interface routines. Only stage two of the protocol compiler will need to be run in order to produce a protocol program in the assembler of the new machine.

Changes in protocol will involve changes to the protocol specification and possibly the program template. All network software will have to be regenerated following the steps outlined above.

7.10. CONCLUSION

This chapter has described the work undertaken to produce a retargetable protocol compiler. It has also described the way this compiler could be used to implement and maintain a computer network.

As can be seen from figure 7.10, the amount of code that has to be hand written on each machine can be made very small. Hence new machines can be added to the network more quickly. This is probably the biggest single advantage of this system. Using this system, major protocol changes can be made much quicker. This is useful since protocol standards can be volatile until they reach maturity.

Any distribution of a compiler based on this design would be enhanced by the provision of a library of i-code routines. In addition, a set of target assembler specifications could be provided. Alternatively, the compiler could be used by a software

house to produce new implementations of standard protocols for their customers. There is clearly much scope for further development of these ideas.

CHAPTER EIGHT

CONCLUSION

8.1. REVIEW

This Thesis has considered the problems of protocol specification and implementation. The introductory chapters discussed basic network principles and described some current practice in the area of protocol specification. The concepts of Wide Area and Local Area Networks were outlined together with some of the techniques used in computer network protocols. Some of the standards which are currently used were briefly discussed.

In the following chapter on protocol specification the two mains types of formal specifications were discussed. These are state transitions specifications and sequence expression specifications. State transition methods include finite state machines and Petri Nets, while sequence expression methods include Calculus of Communicating Systems. Temporal logic was also discussed. State transition methods are more established than sequence expression methods and look to remain so for some time to come. The work of ISO and IBM has produced two protocol specification languages based on one particular state transition approach, the finite state machine. These languages, called ESTELLE and FAPL respectively, have been used in protocol implementation.

Following this background material an alternative approach to protocol specification was described. The data structures of the protocol messages are central to this approach. This differs from the usual approach which is centred on the flow of control within the protocol entities. The feasibility of this approach was explored by developing a protocol modeling system to predict protocol performance. The resulting system proved to be too slow for extensive use. This was due to the characteristics of the host operating system rather than any deficiences in the overall approach.

In the following chapter the problem of the small scale communications user were discussed. The user who only requires inter-host communication occasionally can not justify expenditure on expensive Local Area Networking equipment. He is therefore forced to use asynchronous connection through V.24 ports using the RS232 interface. Various protocols have been devised to run over

asynchronous connections. These include the Kermit and ATS protocols described in this thesis. The requirements for a protocol for the Clearway Network run by the Computer Studies Department at Loughborough University were presented and a general framework for such a protocol was given.

Existing practice in the realm of protocol implementation was then discussed. The problems encountered by implementors of multi-vendor networks were discussed. These include differences in machine architecture, operating system and assembly languages. High-level languages can sometimes be used for protocol implementation, but problems can arise even when using supposedly portable languages such as C. A retargetable protocol compiler was suggested as a possible step forward in this area.

The penultimate chapter describes the design and implementation of a retargetable protocol compiler. The protocol language used in this work differed from previous work in that it did not employ an existing high-level language to specify protocol actions. Hence, the constructs used could be more application specific which enables a more concise description of a protocol than would be otherwise possible. The language design was such that it was fairly simple to produce a protocol compiler, and this compiler has produced code for several target computers. The problems encountered by the protocol implementor are also encountered by the designer of a protocol compiler. The main difficulty lies in producing a general program structures for the program entity. A choice of program structures is available under the protocol compiler.

8.2. FURTHER WORK

Further work is required to verify that the approach adopted in this thesis is entirely practical. The program templates need further development and output programs need to be tested across the Clearway Network.

One possible future application for this work would be automated production of new versions of the Kermit file transfer program. If the specification language could be used to specify the Kermit protocol and suitable program templates developed new

versions of Kermit could be produced quicker and more easily.

8.3. FINAL REMARKS

Although the protocol compiler described in this thesis should be considered as a prototype version, it is has been shown that the principles used in the design are worthy of further consideration. If this system proves to be practical the result would be more accurate and less costly protocol implementation, which would be of great benefit to the networking community. It is hoped that this thesis will stimulate further work in this area.

APPENDIX

SYNTAX OF SPECIFICATION AND INTERMEDIATE LANGUAGES

1.1. SYMBOLS AND ABBREVIATIONS

The meta-language used in this document to specify the syntax of the various specification and intermediate languages is based on Backus-Naur Form. The meaning of the various meta-symbols is defined in the following table.

Meta-symbol	Meaning
=	shall be defined to be
|	alternatively
.	end of definition
[x]	0 or 1 instance of x
{x}	0 or more instance of x
(x|y)	grouping: either x or y
"xyx"	the terminal symbol xyz
meta_identifier	a non-terminal symbol

A meta-identifier shall be a sequence of letters and underscores beginning with a letter.

A sequence of terminals and non-terminal symbols in a production impiles the concatenation of the text they ultimately represent.

1.2. PSL/1

The terminal symbols :

field_ident
integer_ident
frame_ident
class_ident

represent elements of disjoint sets of identifers.
An identifier is an alphanumeric string beginning with a letter.

The terminal symbol:

field_const

is defined as a sequence of ones and zeros enclosed in double quotes.

The terminal symbol

integer_constant

is an integer number.

Specification body

```
specification   =   "protocol" protocol_name
                    parameters
                    state
                    class
                    { class }
                    timer_out_action
                    { timer_action } .
```

Parameters

```
parameters      =   "parameters" "{"
                            param_def
                            { param_def }
                    "}" .

param_def       =   param_name ":=" integer_constant ";" .

param_name      =   ( "send_window"
                    | "receive_window"
                    | "retran_interval"
                    | "timer interval"
```

```
                    ) .
```

State declaration

```
state           =   "state" "{"
                    variable_def
                    { variable_def }
                    "}" .

variable_def    =   ( field_def ";"
                    | integer_def ";"
                    ) .

field_def       =   ( field_ident ":=" field_const
                    | field_ident "[" integer_const "]"
                    ) .
integer_def     =   integer_ident ":=" integer_const .
```

Class declaration

```
class           =   "class" class_ident [frame_type] "{"
                            format
                            frame
                            { frame }
                    "}" .
```

Frame declaration

```
frame           =   "frame" frame_ident [ frame_type ] "{"
                            refinement
                            { refinement }
                            action
                            { action }
                    "}" .

frame_type      =   ( "direct" | "windowed" ) .
```

Format declaration

```
format          =   format "{"
                            frame_field_def ";"
                            { frame_field_def ";" }
                    "}" .
```

```
frame_field_def =       ( field_const
                        | field_ident "[" integer_const "]" [ "frame_id" ]
                        ) .
```

Refinement declaration

```
refinement      =       field_ident "=" format .
```

Actions

```
action          =       "action" "(" act_type ")" "{"
                                act_body
                        "}" .

act_type        =       ( "send" | "receive" | "retran" ) .

act_body        =       primitive_action ";"
                        { primitive_action ";" } .

primitive_action =      ( "if" condition
                          "then" actbody
                          ["else" actbody ]
                          "fi"
                        | "send" "(" frame_ident ")"
                        | "accept"
                        | "cancel" [ "(" range ")" ]
                        | "retran" [ "(" range ")" ]
                        | "start_timer"
                        | "stop_timer"
                        | assignment
                        | "inc" "(" field_ident ")"
                        | "dec" "(" field_ident ")"
                        ) .

condition       =       ( field_ident field_lop field_ident
                        | integer_exp int_lop   integer_exp
                        ).

field_lop       =       ( "=" | "<>" ).

int_lop         =       ( "=" | "<>" | "<" | ">" | "<=" | ">=" ) .

integer_exp     =       ( integer_exp int_op integer_exp
                        | "-" integer_exp
```

```
                    | "-" integer_exp
                    | "(" integer_exp ")"
                    | integer_ident
                    | integer_const
                    ) .

int_op          =   ( "+" | "-" | "*" | "/" ) .

range           =   ( field_ident
                    | field_ident "," field_ident
                    ) .

assignment      =   ( field_ident ":=" field_ident
                    | field_ident ":=" "data"
                    | integer ident := integer_exp
                    ) .
```

Timeout action

```
time_out_action =   "on_timeout" "{"
                          time_out_opt
                    "}" .

time_out_opt    =   ( "single_retran" ";"
                    | "multiple_retran" ";"
                    ) .
```

Timer action

```
timer_action    =   "on_timer_expired" "{"
                          act_body
                    "}" .
```

1.3. PSL/2

The terminal symbols:

seq_field_ident

flag_ident

state_ident

represent elements of disjoint sets of identifiers.
An identifier is an alphanumeric string beginning with a letter.

The terminal symbol

char_const

is defined as an ascii character code enclosed in single quotes.
Mnemonics for unprintable characters are written in upper case.

State declarations

```
state_declarations =    variable_list
                        {variable_list} .

variable_list      =    ( "seq_field" seq_field_list
                        | "flag" flag_list
                        | "state" state_list
                        ) .

seq_field_list     =    ( seq_field_ident "," seq_field_list
                        | seq_field_list
                        ) .

flag_list          =    ( flag_ident "," flag_list
                        | flag_ident
                        ) .

state_list         =    ( state_ident "," state_list
                        | state_ident
                        ) .
```

Frame declarations

```
frame_declarations =    frame
                        { frame } .
```

```
frame              =   frame_type "frame" frame_ident
                               "id_field" char_const
                               field_declaration
                               {field_declaration}
                       "end_frame" .

frame_type         =   ( "windowed" | "direct" ) .

field_declaration  =   ( "seq_field" rule
                       | "data"
                       | "field" char_const
                       | "check_sum"
                       | "params"
                       ) .

rule               =   ( seq_field_ident
                       | "on_send" seq_field_ident
                         "on_receipt" seq_field_ident
                       | "on_receipt" seq_field_ident
                         "on_send" seq_field_ident
                       ) .
```

Events

```
events             =   "events"
                               "start_up" [act_body]
                               "on_open_request" [act_body]
                               "on_close_request" [act_body]
                               "on_character_above" [act_body]
                               "on_character_below" receive_action
                                                    {receive_action}
                               "on_timer_expired" [act_body]
                       "end_events" .

receive_action     =   "[" frame_ident "]:" act_body .

act_body           =   primitive_action
                       { primative_action } .
```

```
primitive_action   =    ( "if" condition
                          "then" act_body
                          ["else" act_body]
                          "fi"
                        | "send_below" frame_ident
                        | "send_above"
                        | "receive"
                        | "discard"
                        | "cancel" range
                        | "retran" range
                        | "start_timer"
                        | "stop_timer"
                        | "enable_above"
                        | "disable_above"
                        | "open_r_window"
                        | "new_state" state_ident
                        | "set" flag_ident
                        | "unset" flag_ident
                        | "inc" field_ident
                        | "dec" field_ident
                        ) .

condition          =    ( condition "or" condition
                        | condition "and" condition
                        | "not" condition
                        | "(" condition ")"
                        | state_ident
                        | flag_ident
                        | seq_field_ident lop seq_field_ident
                        | "s_window_full"
                        | "in_r_window"
                        ) .

lop                =    ( "=" | "<>" ).

range              =    seq_field_ident seq_field_ident .
```

1.4. I-CODE AND TARGET ASSEMBLER SPECIFICATIONS

1.4.1. BASIC NON-TERMINALS

```
type      = ( "char" | "seq" | "addr" | "int" ).

mode      = ( "variable" | "inx_use" | "inx_offset"
            | "inx"      | "param"   | "const"
            ) .

qualifier = ( "eq" | "ne" | "lt" |"gt" |"le" |"ge" | "true" | "false" ).

operator  = ( alloc_op
            | a_2_op
            | a_1_op
            | cntl_op
            ) .

alloc_op  = ( "array" | "variable") .

a_2_op    = ( "mov" | "add" | "sub" | "cmp" | "gad" ) .

a_1_op    = ( "inc" | "dec" | "clr" | "arg" ) .

marker    = ( "beg" | "data" | "text" | "endf" ) .

cntl_op   = ( marker
            | "call"
            | "callc"
            | "jp"
            | "ret"
            | "retc"
            ) .
```

1.4.2. I-CODE AND THE FIVE ELEMENT TUPLE

Five element tuple

The terminal symbol

identifier

is an element of the set of alphanumeric strings which begin with a letter and are less than seven characters in length.

```
The terminal symbols
    length
    offset
    value
are integer numbers.


tuple = tuple_op (type|qualifier) (mode|length) identifier (value|offset).

tuple_op  = ( operator | "lab" ) .
```

I-code

```
operand     =     ( identifier
                  | [cast] [offset] "[inx]"
                  | [cast] [offset] "[param]"
                  | [cast] integer
                  | char_const
                  | inx
                  ) .

cast        =     "(" type ")" .

program     =     line
                  { line } .

line        =     ( label
                  | instruction
                  ).

instruction =     ( alloc_ins
                  | a_2_ins
                  | a_1_ins
                  | cntl_ins
                  ) .
```

Allocation instructions

```
alloc_ins   =     ("array" type length identifier
                  |"variable" type identifier value
                  ) .
```

Arithmetic two operand instructions

```
a_2_ins     =    a_2_op operand "," operand .
```

Arithmetic one operand instructions

```
a_ins       =    a_1_op operand .
```

Control instructions

```
cntl_ins    =    ( marker
                 | "call" identifier value
                 | "callc" identifier value
                 | "jp" [ qualifier ] identifier
                 | "ret"
                 | "retc" ( "true" | "false" )
                 ) .
```

1.4.3. TARGET ASSEMBLER SPECIFICATION

A template is a string of characters surrounded by double quotes as described in the main text.

```
spec        =  { definition }
               { ( table | single_template) } .

comp_type   =  ( type | identifier ) { "|" ( type | identifier ) } .

comp_mode   =  ( mode | identifier ) { "|" ( mode | identifier ) } .

definition  =  ( "def_type" identifier "=" comp_type
               | "def_mode" identifier "=" comp_mode
               ) .

table       =  ( "table" "alloc"
                       { alloc_op "," comp_type "," template }
                 "end_table" .
               | "table" "arith_two_op"
                       { a_2_op "," comp_type "," comp_mode ","
                         comp_type "," comp_mode "," template }
                 "end_table"
               | "table" "arith_one_op"
                       { a_1_op "," comp_type "," comp_mode ","
                         template }
                 "end_table"
               | "table" "jp"
                       { qualifier "," template }
                 "end_table"
               | "table" "retc"
                       { ( "true" | "false" ) "," template }
                 "end_table"
               | "table" "subs"
                       { mode "," template "," template }
                 "end_table"
               ) .

single_temp =  "template" temp_name template .
```

```
temp_name   =  ( "lab_use"
               | "lab_dcl"
               | "beg"
               | "call"
               | "callc"
               | "ret"
               | "text"
               | "data"
               | "endf"
               ) .
```

REFERENCES

Abbreviations :-

ACM	- Association for Computer Machinery.
CACM	- Communications of ACM.
ACM Toplas	- ACM Transactions on Programming Languages and Systems.
ICC	- International Conference on Communications.
IEEE	- Institute of Electical and Electronics Enginneers.
IEEE Trans. Comm.	- IEEE Transactions on Communication.
IFIP	- International Federation for Information Processing.

ALFONZETTI S., 1982, From Formalism to Implementation of OSI Entities, 15th International Conference on System Sciences, Hawaii, Jan 1982, pp 414-442.

AYACHE J.M., COURTIAT J.P., DIAZ M., 1982, A Specification Language for the Design of Multi-Layer Protocols, 15th International Conference on System Sciences, Hawaii, Jan 1982, pp 404-413.

BAUERFIELD W.L., 1983, Performance Prediction of Computer Network Protocols, Proc. ICC 1983 Boston, Mass., June 1983, E4.5.1-E.4.5.5.

BIDMEAD C., 1982, Review, Practical Computing, September 1982, pp 61-64.

BLUMER P.T. & R.L.TENNEY, 1982, A Formal Specification Technique and Implementation Method for Protocols, Computer Networks Vol 6., July 1982, pp 201-217.

BOCHMAN G.V. & CHUNG R.J., 1977a, A Formal Specification of HDLC Classes of Procedures, Proc. National Telecommunications Conference, 1977, Reprinted in **CHU W.W., 1979,** Advances in Computer Communication and Networking, Dedham, Mass, Artech House, pp 519-530.

BOCHMANN G.V. & GECSEI J., 1977b, A Unified Method for the Specification and Verification of Protocols, Proc. IFIP. Congress 1977, Toronoto, pp 229-234.

BOCHMANN G.V. & SUNSHINE C.A., 1980, Formal Methods in Communication Protocol Design, IEEE Trans. Comm. Vol COM-28, April 1980, pp 624-631.

BROWN K., Linguistics Today, Fontana.

BT, 1983, Networking over Asynchronous lines, Prepared by the British Telecom New Networks Technical Forum, Feb 1983.

CATTELL R.G.G., 1980, Automatic Derivation of Code Generators from Machine Descriptions, ACM Toplas, Vol 2. No.2, April 1980, pp 173-190.

CLARK D.D., POGRAN K.T. and REED D.P., 1978, An Introduction to Local Area Networks, Proc IEEE Vol. 66, No. 11, Nov 1978, pp 1497-1517.

COLEMAN S.S., POOLE P.C. & WAITE W.M., 1974, The Mobile Programming System, JANUS, Software, Practice & Experience, Vol 4, 1974, pp 5-23.

COOKE D.J. & BEZ H.E., 1984 Computer Mathematics, Cambridge University Press, 1984, pp 321-328.

DA CRUZ F. & CATCH B., 1984 Kermit: A File-Transfer Protocol for Universities, Byte, No. 6, June 1984, pp 255-278, and No. 7, July 1984, pp 143-145 & 400-403.

DANTHINE A.A.S, 1980, Protocol Representation with Finite State Models, IEEE Trans. Comm., Vol. COM-28, April 1980, pp 632-643.

DAVIES D.W., BARBER D.L.A., PRICE W.L. & SOLOMONES C.M., 1979, Computer Networks and their Protocols, John Wiley & Sons Ltd, 1979.

DIAZ M., 1982, Modeling and Analysis of Communication and Co-operation Protocols using Petri Net Based Models, Computer Networks, Vol. 6, No. 6, Dec 1982, pp 419.

DP8807, 1985, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observation Behaviour, ISO Draft Proposal 8807, March 1885.

EHRIG H., FREY W. & HANSEN H., ACT ONE: An Algebraic Specification Language with two levels of semantics, Bericht Nr 83-03, Technische Universitaet Berlin.

FIELD J.A., 1976, Efficient Computer-Computer Communication, Proc. IEE vol. 123, Aug 1976, pp 756-760.

FRASER A.G., 1977, Delay and Error Control In Packet Switched Network Proc. ICC 1977, p 22.4-121 - 22.4-125.

GRANVILLE R.S. & GRAHAM S.L., 1978, A New Method of Compiler Code Generation, Proc. 5th Conf. on Principles of Programming Languages, Jan 1978, pp 231-240.

HAILPERN B.T. & OWICKI S., 1983, Modular Verfification of Computer Communication Protocols, IEEE Trans. Comm., Vol COM-31, No. 1, Jan 1983, pp 56-69.

HARANGOZO J., 1977 An Approach to Describing a Link Level Protocol with a Formal Language, Proc. 5th Data Comm. Symposium. Utah, 1977, pp 4-37 to 4-49.

JOHNSON S.C., 1978a, A Tour through the Portable C Compiler, Unix Programmers Manual, 7th Edition, Vol 2, 1978.

JOHNSON S.C., 1978b, Yacc: Yet Another Compiler-Compiler, Unix Programmers Manual, 7th Edition, Vol 2, 1978.

JONES C., 1980, Software Development - A Rigorous Approach, Prentice-Hall.

KELLER R.M., 1969, Formal Verification of Parallel Programs, CACM Vol. 12, No. 7, 1969, pp 371-384.

KERNIGHAN B.W. & RITCHIE D.M., 1978, The M4 Macro Processor, Unix Programmers Manual, 7th Edition, Vol 2, 1978.

LAMPORT L., 1983, Specifying Concurrent Program Modules, ACM Toplas, Vol 5., No. 2., April 1983, pp 190-222.

LARMOUTH J., 1982, Cambridge Ring 82 - Protocol Specification, Joint Network Team of the Computer Board and Research Councils, Nov 1982.

LE LANN G. & LE GOFF H., 1978, Verification & Evaluation of Communication Protocols, Computer Networks, Vol 2, Feb 1978, pp 50-60.

LESK M.E. & SCHMIT E., 1978, Lex - A lexical Analyser Generator, Unix Programmers Manual, 7th Edition, Vol 2, 1978.

MERLIN P.M., 1979 Specification and Validation of Protocols, IEEE Trans. Comm., Vol COM-27, Nov 1979, pp 1671-1680.

METCALF R.M. & BOGGS D.R., 1976, Ethernet: Distributed Packet Switching for Local Computer Networks, CACM Vol 19, No 7, July 1976, pp 395-404.

MILNER R., 1980, A Calculus of Communicating Systems, Lecture Notes in Computer Science, No 92, Springer-Verlag, 1980.

NASH S.C., 1983, Automated Implementation of SNA Communication Protocols, Proc. ICC 83, Boston Mass., June 1983, pp 1316-1322.

PETERSON J.L., 1977, Petri Nets, ACM Computer Surveys, Sept 1977, pp 223-252.

PIATKOWSKI T.F., 1983, Protocol Engineering, Proc. ICC 83, Boston Mass., June 1983, E4.8.1-E4.8.5.

POOLE P.C., 1974, Portable & Adaptable Compilers, In Compiler Construction - An Advanced Course, Lecture Notes in Computer Science, Vol. 21, Springer-Verlag, New York 1974, pp 427-497.

POZEFSKY D.P. & SMITH F.D., 1982, A Meta-Implementation for Systems Network Arcitecture, IEEE Trans. Comm., Vol COM-30, No. 6, June 1982, pp 1348-1355.

REISER M., 1982, Performance Evaluation of Data Communication Systems, Proc. IEEE, Vol. 70, No. 2, 1982.

RTDL, 1984, Clearway User Guide, Issue 7, Real Time Developments Ltd., October 1984.

SALTZER J.N. & POGRAN K.T., 1979, A Star Shaped Ring Network with High Maintainability, Proc. Local Area Communication Network Symposium, Mitre Corp., Boston, May 1979, pp 179-190.

SCHULTZ G.D., ROSE D.B., WEST C.H. & GRAY J.P., Executable Description and Validation of SNA, IEEE Trans. Comm., Vol. COM-28, No. 4, April 1980, pp 661-676.

SCHWART R.L. & P.M. MELLIAR-SMITH, 1982, From Finite State Machines to Temporal Logic, IEEE Trans. Comm., Vol. COM-30, No. 12, Dec 1982.

STENNING N.V., 1979, Definition and Verification of Computer Network Protocols, NPL Report DNACS 15/78, Feb 1979.

SUNSHINE C.A., 1978, Survey of Protocol Definition and Verification Techniques, Computer Networks, Vol 2, No 4/5, Sept/Oct 1978, pp 346-350.

SUNSHINE C.A., 1979, Formal Techniques for Protocol Specification and Verification, Computer, Vol. 12, Sept 1979, pp 20-27.

TANENBAUM A.S., 1981, Computer Networks, Prentice-Hall Inc., 1981.

TENNEY R.L., 1983, One Formal Description Technique for ISO OSI, Proc. ICC 83, Boston, Mass., June 1983,

WAITE W.M.. 1970, The Mobile Programming System: STAGE2, CACM Vol. 13, July 1970, pp 415-421.

WILKES M.V. & WHEELER D.J., 1979, The Cambridge Digital Communication Ring, Proc. Local Area Communication Network Symposium, Mitre Corp, Boston, May 1979, pp 47-60.

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