University of Natal

# Using a Terminal Switching Exchange for Computer Peripheral Sharing and Other Purposes

A. P. Barrett

University of Natal

# Using a Terminal Switching Exchange for Computer Peripheral Sharing and Other Purposes

Alan Peter BARRETT, B.Sc.Eng.

Submitted in partial fulfilment of the requirements for the degree of Master of Science in Engineering, in the Department of Electronic Engineering, University of Natal, Durban.

March 1988

## Abstract

Several computers and several tens of terminals at the University of Natal are linked to a terminal switching exchange network known as NUNET, which is in fact part of a country-wide network known as NIINET. This thesis describes software that was written to enhance the usefulness of the network, particularly in the Department of Electronic Engineering.

The enhancements include a simple facility to provide help to users of the network, and programs that allow various computers in the Department of Electronic Engineering to create and close network connections. The programs that create and close network connections are used to provide access to printers and plotters that are connected to the network and shared by several computers. Access to peripherals through the network has been integrated into existing printer and plotter driver programs on some of the computers, thus allowing the network to be completely transparent to the user. The same network access programs also allow connections to be made between computers for the purpose of file transfers.

## Preface

The work described in this thesis was performed in the Department of Electronic Engineering at the University of Natal, in 1985, 1986 and 1987. The work was initially under the supervision of Mr D.C. Levy, and later under the supervision of Mr R. Peplow.

I thank my supervisors for their guidance and encouragement, Mr M.D. Spann for providing some useful suggestions, and my colleagues for contributing to a pleasant working environment. I am grateful to the CSIR and to the University of Natal for their financial support.

I certify that all the material incorporated in this thesis is my own unaided work, except where specific acknowledgement is made. The work contained herein has not been submitted for a degree at any other university.

Signed: [illegible]

A.P. Barrett
Durban, March 1988

## Table of Contents

## Glossary

| | |
|---|---|
| ABR | Automatic Baud-rate Recognition. |
| AMP | Automatically Mapped Port. A type of port on the CASE network that will be automatically connected to a suitable destination when a connect event occurs. |
| ANSI | American National Standards Institute. |
| ARQ | Automatic Repeat Request. A protocol used on the high speed composite links in the CASE network. |
| ASCII | American Standard Code for Information Interchange. A standard alphabet used by computers. |
| Baud | Number of low level symbols transmitted per second on a communications channel. If the low level symbols are bits, then a channel's baud rate is the same as the number of bits per second. |
| Bps | Bits per second. |
| Break | An signal, consisting of a long space condition, sent on an asynchronous communications line. |
| CASE | Computer And Systems Engineering plc. The manufacturer of the equipment used in a CASE network. |
| CASE Network | A network of DCX-850 nodes. |
| CCITT | International Telegraph and Telephone Consultative Committee. |
| Concentrator | Statistical multiplexer. |
| CTS | Clear To Send. One of the RS-232-C handshake lines. |
| DCD | Data Carrier Detect. One of the RS-232-C handshake lines. |
| DCE | Data Communications Equipment. Equipment that allows DTE'a to communicate with one another. |
| DCX-815 | A statistical multiplexer, manufactured by CASE, used at the University of Natal. |
| DCX-840 | A terminal switching exchange *cum* statistical multiplexer, manufactured by CASE, without the ability for users to initiate their own connections. |
| DCX-850 | Same as a DCX-840, but with the ability for users to initiate and terminate connections. |
| DTE | Data Terminal Equipment. A device such as a computer or terminal. |
| DTR | Data Terminal Ready. One of the RS-232-C handshake lines. |
| EENET | Electronic Engineering Network. Actually node 8 of NUNET. |
| EIA | Electronic Industries Association. |
| Flow Control | Control over data flow between devices, to prevent buffer overflows when data arrives too fast to be processed. |
| HDLC | High Level Data Link Control. Used on high speed composite links in the CASE network. |
| IMP | Internally Mapped Port. A type of port on the CASE network that is permanently connected to a particular destination port. |
| ISO | International Standards Organisation. |

| | |
|---|---|
| Kermit | A file transfer protocol, or a program that implements this protocol. |
| LSC | Low Speed Channel, especially on the CASE network. |
| LU | Logical Unit. A number that identifies a peripheral device to a computer operating system. |
| Modem | MOdulator/DEModulator. |
| MUX | Multiplexer. |
| NIINET | National Institute of Informatics Network. A country-wide network of CASE DCX-850 nodes. |
| NUNET | Natal University Network. Nodes 8, 9 and 10 of NIINET. |
| OSI | Open Systems Interconnection. |
| RS-232-C | A standard physical layer interface for serial computer communications. |
| RTS | Request To Send. One of the RS-232-C handshake lines. |
| Shell | A command interpreter, especially under the Unix operating system. |
| Statistical Multiplexer | A terminal multiplexer that uses statistical methods instead of fixed time division multiplexing. |
| Superuser | A computer user granted extra power by the operating system, in addition to the capabilities of an ordinary user. |
| Terminal Emulator | A program that allows a computer (usually a personal computer) to function as a terminal, for connecting to other computers. |
| Terminal Multiplexer | A device that allows several low speed serial links to be multiplexed onto a single, high speed, link. |
| UMP | User Mapped Port. A type of port on the CASE network that allows a user to select a destination to which he wishes to connect. |
| XOFF | A character transmitted by one device in order to request another device to cease sending data. |
| XON | A character sent by one device to inform another device that it may resume sending data. |
| XON/XOFF | A flow control method employing XON and XOFF characters. |

## Section 1. Introduction

When a computer system supports several users simultaneously, each user generally makes use of a terminal to interact with the computer. In the simplest cases, these terminals are connected directly to the host computer by serial communication lines. Similar serial communication lines are often used for communications between computers and peripheral devices other than terminals, such as printers or plotters.

Switching arrangements of various kinds are often used between terminals and computers. A *terminal switching exchange* [Hals85], for example, allows a terminal user to choose to connect his terminal to one of several computers. This connection is made after a user-initiated dialogue between the user and the exchange. A connection made in this way can usually be terminated by either the terminal user or the computer. The terminal switching exchange thus resembles the public telephone system.

Computers and their peripheral devices may be linked to form a *resource sharing network* (often simply called a *network*), which is a system that allows a user at one location to access resources that are located elsewhere. Such a network usually consists of a collection of interconnected computers and peripheral devices. Resource sharing networks fall into two major categories: *Computer Networks* and *Computer-Communications Networks* [Elov74]. A computer-communications network appears to the user to be a collection of several distinct computing systems, which the user must manage explicitly. A true computer network appears to the user to be a single large computing system, whose detailed organisation is irrelevant to the user. Some networks exhibit characteristics of both classes.

In the Department of Electronic Engineering at the University of Natal, there are several tens of terminals and personal computers (PCs), and several minicomputers and other devices. Many more terminals and several other computers are located elsewhere on both the Durban and Pietermaritzburg campuses of the University. Many of these computers and terminals, and some other devices, have serial communications lines connected to *NUNET*, which is a network of terminal switching exchanges but which appears to the user to be a single terminal switching exchange. NUNET is often referred to simply as a *CASE* network, after the manufacturer of the switching equipment used in it.

The CASE network makes it very easy for a terminal user to request that his terminal be connected to a particular computer, but does not readily support the functions required of a true computer network, as outlined above.

It was desired to enhance the usefulness of the CASE network in the Department of Electronic Engineering at the University of Natal. There were three major objectives in this regard: to provide a facility enabling users of the network to obtain help and news about the network; to allow peripheral devices such as printers and plotters to be shared among several computers; and to allow files to be transferred between computers connected to the network.

The following minicomputers operated by the Department of Electronic Engineering are connected to the CASE network:

A Hewlett-Packard HP-1000F, with the RTE-6 operating system.

A Gerber IDS-80, with the DMS-2 operating system (which is based on HP's RTE-4).

A Hewlett-Packard HP-1000/A900 (usually referred to simply as an HP-A900), with the RTE-A operating system.

A Hewlett-Packard HP-9000 model 500, with the HP-UX operating system (HP's version of Unix).

A DEC VAX 11/750 (usually referred to simply as a VAX-750), with the VMS operating system.

The remainder of this thesis describes enhancements made to the CASE network by the author. The switching equipment that forms the network was not modified; the enhancements are in the form of software that runs on the computers connected to the network.

## Section 2. Networking Overview

This section presents an overview of computer communications and networks. Subsection 2.1 deals with connections between computers and serial peripheral devices such as terminals, including terminal multiplexers and terminal switching exchanges. Subsection 2.2 deals with connections between computers, in various types of networks. Subsection 2.3 discusses layered models for computer networks.

### 2.1. Connections Between Computers and Peripherals

In order for two devices (often a computer and a peripheral device used by the computer) to exchange information, they must agree on several conventions. Some of the issues that must be addressed are the numeric codes used to represent character data, the electrical characteristics of the link, and methods of flow control or handshake to prevent a slow device from being swamped by information from a faster device. In addition to these considerations, it is of course essential that electrical signals transmitted by one device are delivered to the other device.

#### 2.1.1. Character Codes

Several schemes exist for encoding characters to be interchanged electronically between communicating devices. The most common character codes used in computers are *EBCDIC* (Extended Binary Coded Decimal Interchange Code), which is an eight-bit code, and *ASCII* (American Standard Code for Information Interchange), which is a seven-bit code. ASCII characters are often transmitted using eight bits, in which case the eighth bit might be set to a fixed value, used as a parity check, or used to support extended (non-ASCII) characters. The ASCII code is also known as USASCII, and is an ANSI standard (ANSI X3.4-1977). The CCITT standard International Alphabet Number 5 (IA5) is equivalent to the ASCII code.

The ASCII code is used almost exclusively by devices connected to NUNET, although it is not used internally by all computers connected to the network. IBM mainframes use EBCDIC internally, and require special front-end processors to allow them to be used with ASCII terminals connected through the network.

ASCII defines codes for upper- and lower-case alphabetic characters, digits, and several punctuation symbols. It also defines several special control codes, whose usage varies widely. A table of ASCII codes appears in Appendix A1.

#### 2.1.2. Serial Communications

Computers often use fairly low speed serial communications lines (running at rates of up to a few thousand bits per second) to communicate with peripheral devices such as terminals, printers and plotters.

Simple serial communication lines do not function well over long distances, because digital signals are subject to distortion. A modulated analogue carrier is better suited to long distance transmission, and *modems* make use of this principle to extend the range over which a serial communication line may be used. The name *modem* stands for *MODulator-DEModulator*, which is a brief description of the modem's task: it modulates digital information onto an analogue carrier prior to transmission, and demodulates the analogue signal, converting it back to a digital signal after reception at the other end of a link.

EIA standard RS-232-C [EIA79] defines an "Interface Between Data Terminal Equipment and Data Communications Equipment Employing Serial Binary Data Interchange". Items covered by this standard include the order in which bits are transmitted on the data lines, the usage of several control lines, and the electrical characteristics of the data and control signal lines. Serial communications lines purporting to comply with this standard are often referred to as *RS-232-C lines*.

This standard was designed for use between computers or terminals (known as *Data Terminal Equipment*, or *DTE*) and modems (*Data Communications Equipment*, or *DCE*), and is compatible with the CCITT V.24 standard. All the signals used in the RS-232-C and V.24 standards are listed in Appendix A2. The usage of the various RS-232-C control signals varies widely, but the signals and meanings listed below are common:

| | |
|---|---|
| RTS | Request to Send. Used by a DTE to request the DCE to enter transmit mode. |
| CTS | Clear to Send. Used by a DCE to tell the DTE when it is permitted to send data. |
| DSR | Data Set Ready. Indicates that the local DCE is ready, but may not be connected to a remote DCE. |
| DTR | Data Terminal Ready. Often used by the DTE to tell the DCE when it is permitted to send data, although this meaning is not provided for in the standard. |
| DCD | Data Carrier Detect. Indicates that the local DCE is in contact with a remote DCE. |

Characters are transmitted bit-serially, with the first transmitted bit being the least significant. Transmission may be synchronous or asynchronous [Lam84]. In synchronous transmission, the data signal lines are never idle, and clocks at the transmitting and receiving interfaces remain synchronised. Special fill characters are inserted if the there is no data to be transmitted, so that the interfaces remain synchronised. In asynchronous transmission, on the other hand, there is an idle time between one character and the next. The receiving interface must re-synchronise its clock at the start of each received character. All the low-speed ports on NUNET use asynchronous transmission, but the composite links internal to the network use synchronous transmission.

Figure 2.1 illustrates the asynchronous transmission of a single eight-bit character. The idle state of the data signal line is the logic 1 level, (which corresponds to a negative voltage). The start of a character is indicated by a *start bit* of logic level 0 (a positive voltage), whose duration is one bit-time. The eight data bits are transmitted in the eight bit-times following the start bit. The least significant bit is the first data bit to be transmitted, and the most significant bit (which may be a parity bit, depending on the application) is the last data bit to be transmitted. After transmission of the last data bit, the line returns to the idle state (logic level 1), where it must remain for a predetermined minimum period, before the next character may be sent. The duration of this minimum idle period is usually 1, 1.5 or 2 bit-times, and a particular configuration is therefore said to use 1, 1.5 or 2 *stop bits*.

When asynchronous transmission is used, a signal known as a *break* can be transmitted. A break signal is not a character. It is transmitted by causing the data line to remain in the space condition (logic level 0) for a period longer than the total time taken to transmit a character. In practice, the duration of the break condition is usually many character times.

Although the RS-232-C standard was designed for use in connections between a DTE and a DCE, it has been applied to other configurations, a very common one being between a computer and a peripheral device that is connected to it directly, without the use of modems. When two DTE's are connected directly in this way, the cable used for the connection is referred to as a *null modem cable*. Such a cable must interchange the transmit and receive data lines between the two ends, and must also see that the necessary control lines are interchanged or looped back in a suitable way. In many installations, the control lines defined in the RS-232-C standard are not all present, and frequently only the data lines and ground line are used, without any of the control lines.

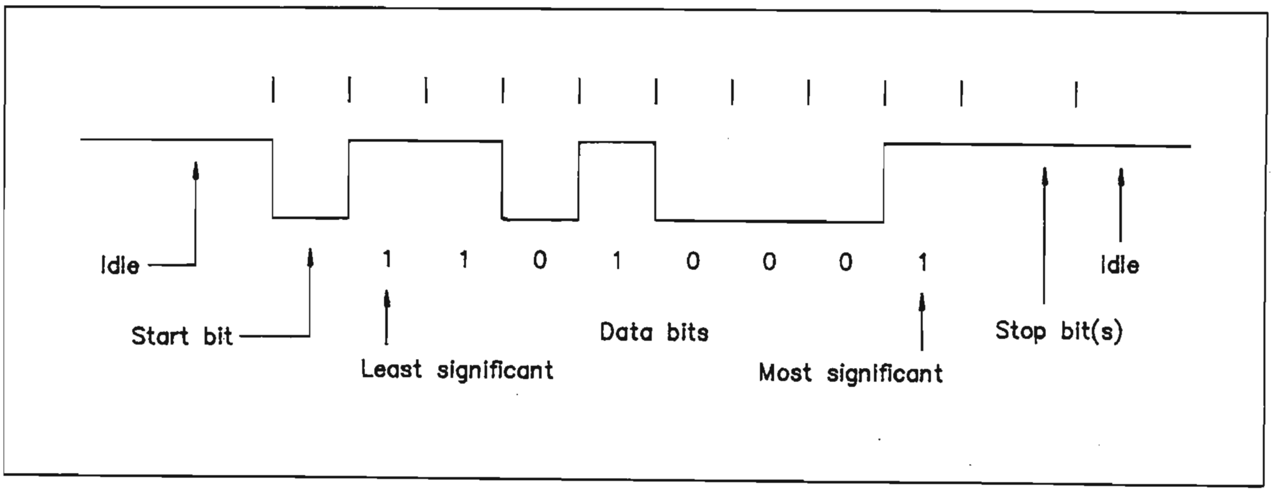

**Figure 2.1** Asynchronous transmission of an eight-bit character.

### 2.1.3. Flow Control in a Serial Link

When two devices communicate using a serial link, it is often necessary for the receiving device to request the transmitting device to stop sending (temporarily), in order to allow relatively slow devices to avoid being swamped by data from faster devices. Even in communications between two similar devices, some form of flow control is often necessary.

Flow control methods fall into two important categories: *in band* and *out of band*. In band flow control uses signals similar to those used for data exchange, while out of band flow control uses signals entirely different from those used for data transmission. In the case of serial RS-232-C lines, in band flow control would involve the transmission of special characters on the primary data lines, while out of band flow control would either use signals transmitted on the secondary data channel or on the special control lines defined in the RS-232-C standard.

Where an installation uses at least some of the RS-232-C control lines, then the device at each side of the link typically uses one of the control lines to inform the other side whether or not it is ready to receive data. This requires two control lines (one in each direction), unless one of the devices can guarantee that it will always be ready to receive data. This form of out-of-band flow control is often referred to as *hardware handshake*. One common, but by no means universal, method of flow control requires the device to assert the *Data Terminal Ready* (DTR) signal (by driving it to a positive voltage) when it is ready to receive data, and checks the *Clear To Send* (CTS) signal for permission to transmit data. Unfortunately, different devices often have different control line usage requirements, which can result in much confusion when a new device is connected to an existing system.

When an installation does not use the RS-232-C control lines, some other form of flow control is usually required to prevent a device from being swamped with more data than it can handle. One common method, which is used by most devices and computers on NUNET, is known as *XON/XOFF flow control*. In this scheme, a device transmits a special character called the *XOFF* character when it wished the other side to cease transmitting data. Another special character called the *XON* character is output to give the other side permission to resume transmission. The *XON* and *XOFF* characters are usually the ASCII *DC1* and *DC3* characters (also known as *control-Q* and *control-S*), respectively.

XON/XOFF flow control may operate in either one or both directions. Operation in only one direction is acceptable when one of the devices is willing to receive data at any time, and will never need to prevent the other device from transmitting. Operation in both directions is required when each device might need to prevent the other device from transmitting.

### 2.1.4. Terminal Multiplexers and Concentrators

A *terminal multiplexer* is a device that allows several low-speed serial lines to be transmitted via a single high-speed connection [Pehr73] [Loom83] [Hals85]. This arrangement allows line costs to be reduced, especially where several remote devices are fairly close to one another but are distant from a facility to which they require access. Figure 2.2 illustrates the use of two terminal multiplexers with a modem link.

The simplest varieties of terminal multiplexers use a fixed time-division multiplexing technique, which divides the capacity of the high speed channel equally among the low speed channels regardless of whether or not, at any particular instant, each low speed channel requires the capacity allocated to it. For example, eight low speed channels at 2400 bps could be time-division multiplexed onto a single high speed channel at 19200 bps.

Statistical multiplexers, also called terminal concentrators [Pehr73] [Loom83], do not use fixed time-division multiplexing. Instead, they divide the capacity of the high speed channel according to the dynamic requirements of the low speed channel. The capacity of the high speed channel is usually less than the combined capacities of the low speed channels. For example, a statistical multiplexer might allow eight channels at 9600 bps to be concentrated onto a single 19200 bps channel. This arrangement would, of course, not permit all the low speed channels to use the full 9600 bps simultaneously. It would, however, allow one channel to be used at the full 9600 bps while several other channels are used with a lower throughput. (The CASE DCX-815 statistical multiplexer, several of which are in use at the University of Natal, multiplexes up to eight channels, of up to 9600 bps each, onto a single 9600 bps or 19200 bps channel.)

Typical computer communications are bursty in nature. For this reason, the aggregate throughput required (at any instant) of the low speed lines connected to a concentrator (or statistical multiplexer) is often within the capacity of the composite link. This often allows concentrators to provide a link with little noticeable performance penalty under normal circumstances, although performance would be degraded when the total (instantaneous) desired throughput on all the low speed channels exceeded the available capacity of the composite link.

Encoding of the state of the RS-232-C control lines on the low speed channels so that their state at one end of the link matches that at the other end is sometimes required of terminal multiplexers or concentrators. This results in some additional overhead.

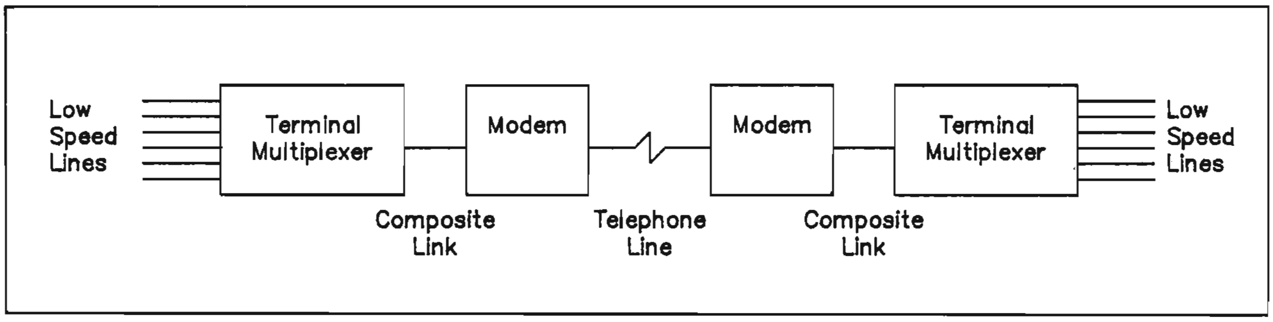

**Figure 2.2** Typical terminal multiplexer or concentrator configuration.

### 2.1.5. Terminal Switching Exchanges

A user may require the ability to connect his terminal to any one of several computers. This facility can be provided by a centralised patch panel, which allows an operator to physically switch plugs from one circuit to another, much as in an old-fashioned manual telephone exchange. Of course, links between computers and terminals are not the only type that could be managed in this way; connections between two computers or between a computer and some other peripheral device could be treated in the same way.

A *terminal switching exchange* [Hals85] may be used to replace a patch panel. The terminal switching exchange has many low speed serial lines connected to it, and effectively connects pairs of low speed channels together, so that input on one channel appears as output on the associated channel. The exchange can change the routing of individual low speed channels at the request of the users connected to the channels. A user at a terminal can get the attention of the exchange and request to have his channel connected to a specified destination channel. This is similar in concept to making a connection on the public telephone network: Getting the attention of the terminal switching exchange corresponds to lifting the handset and waiting for a dial tone; Requesting to be connected to a particular destination corresponds to dialing a telephone number. Figure 2.3 illustrates the use of a terminal switching exchange to allow terminal users to connect to either of two computers.

Simpler types of terminal switching exchange might require intervention by an operator to perform this switching function. Such exchanges do not satisfy the definition given in [Hals85], and are equivalent in function to a patch panel that allows an operator to connect one port to any other.

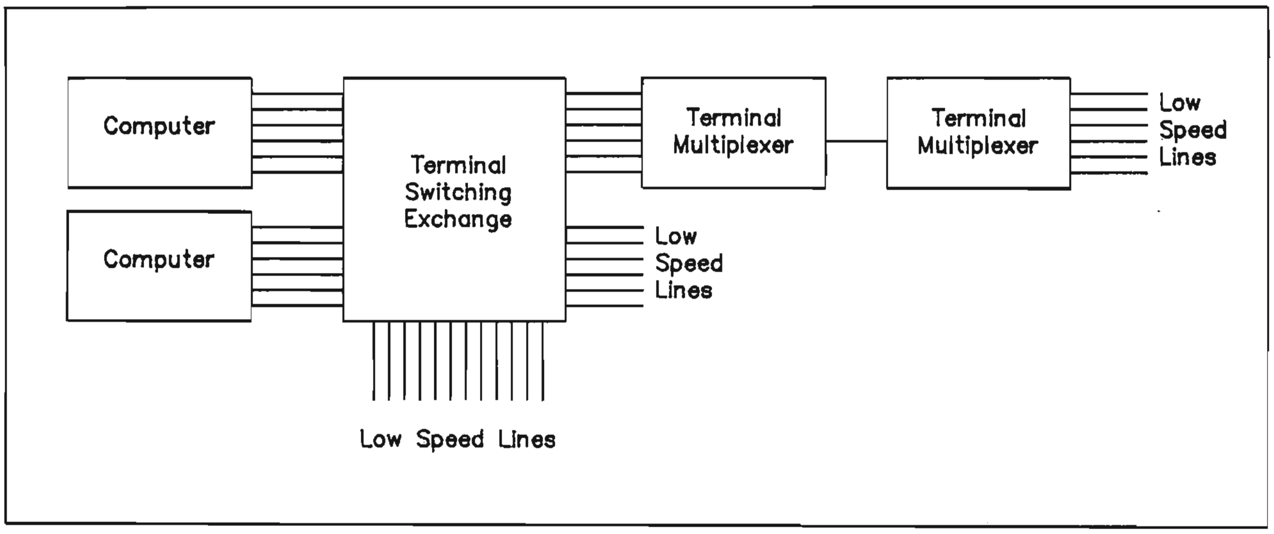

**Figure 2.3** A terminal switching exchange. Terminals connected to the low speed ports can access either computer, on request.

## 2.2. Networks

A communications network must be capable of passing information from any station to any other station on the network. When the number of devices attached to the network is large, it is not feasible to provide a dedicated connection between each device and every other device, because the number of connections required in such a fully connected network is approximately proportional to the square of the number of nodes.

Two important classes of communications networks address this interconnection problem in very different ways. A *broadcast* network allows stations connected to it to transmit information which is received by many or all other stations on the network. A *switching* network contains elements that allow a signal from one station to be switched among a variety of possible pathways, until it eventually arrives at the taken by a destination station.

Networks that employ broadcast techniques include terrestrial packet radio (such as the University of Hawaii's ALOHA network), satellite networks, and local area networks (including both CSMA networks such as Ethernet, developed jointly by Digital Equipment Corporation, Intel and Xerox, and the IEEE 802.3 standard, as well as token passing bus networks such as the IEEE 802.4 standard).

Networks that employ switching techniques include the public telephone system, public packet switched networks (such as the South African Post Office's SAPONET), networks provided by computer manufacturers (such as IBM's SNA and DEC's DECNET), and wide area computer communications networks (such as ARPANET).

Switching networks can be divided into three classes: *circuit switching, message switching* and *packet switching* networks.

A circuit switching network establishes a direct link between the two communicating stations before any communication can take place. Once the connection has been established, communication between the two stations proceeds as if they were directly connected by a physical line. At the end of the conversation, the circuit is disconnected and the network resources that made up the circuit become available for another connection. The public telephone network is a circuit switching network; when a caller dials a number, a circuit is set up between the caller's telephone and the receiver's telephone, conversation between the two parties then continues as if they were directly connected, until one of the parties terminates the call.

A message switching network uses *store and forward* techniques to transport messages from a source station to a destination station. A message must often travel through several intermediate network nodes on its journey from the source to the destination station. Each network node will receive the entire message, storing it in buffer memory, before re-transmitting the message on the next stage of its journey. When the messages are large, the time elapsed between transmission of a message by the source station and reception by the destination station can be substantial. This is not a problem for facilities such as electronic mail, but is clearly unacceptable for voice traffic.

A packet switching network is similar to a message switching network, except that messages are broken up into fairly small *packets* before transmission. When a message needs to pass through several network nodes on its way to the destination station, each intermediate node will receive a packet in its entirety before forwarding that packet to the next node. Packet switching networks may provide either a *datagram* or a *virtual circuit* service.

In a datagram service, each packet is routed through the network independently, and there is no guarantee that the packets will arrive in any particular order, because several packets belonging to the same message might take different routes through the network, depending on the routing strategy followed by the network.

In a virtual circuit packet switching service, there is an initial call setup procedure, during which network resources required to support the call are allocated. After the initial setup, packets belonging to that call are guaranteed to arrive in the correct order, and their processing by the network may be more efficient than the processing of individual datagram packets.

Figure 2.4 [Stal85] illustrates the four major types of switching methods (circuit switching, message switching, datagram packet switching and virtual circuit packet switching) for a configuration in which a message must pass through two intermediate nodes before reaching its destination.

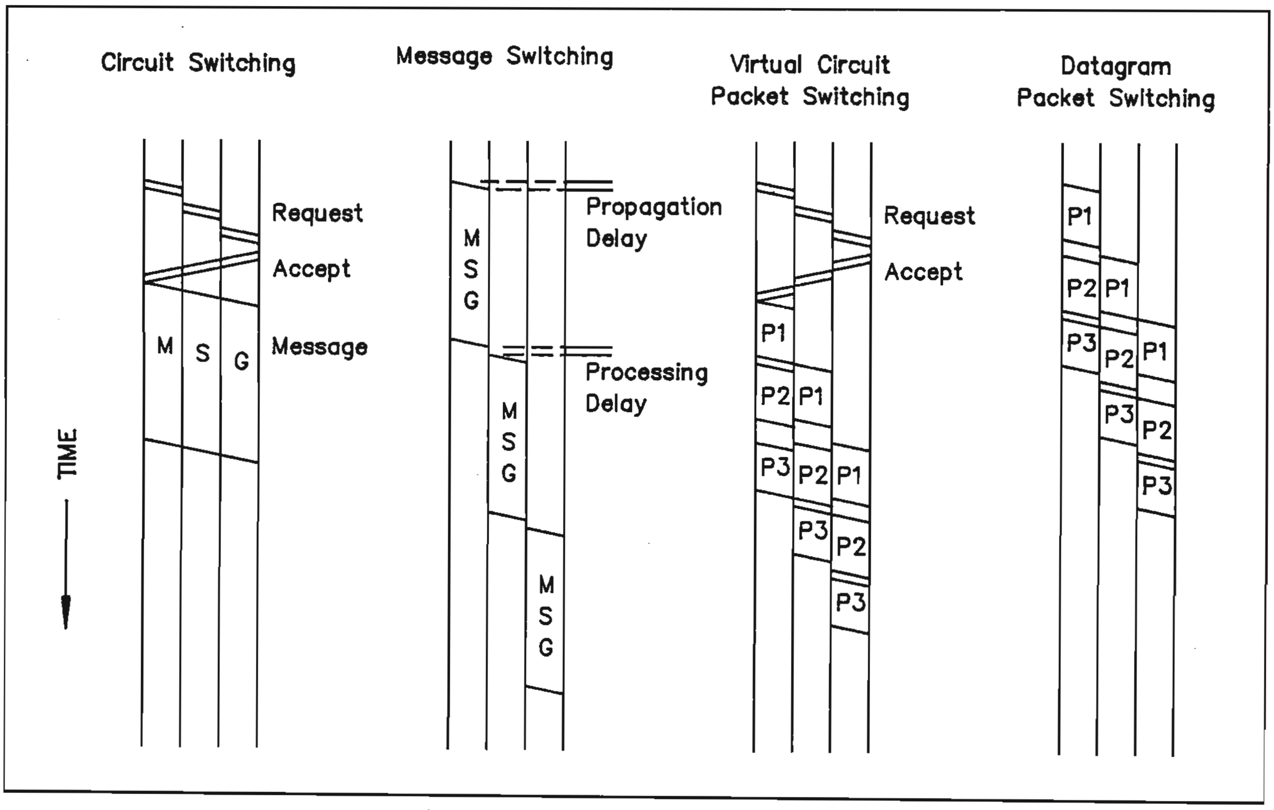

**Figure 2.4** Event timing for various switching techniques, where the source and destination are separated by two intermediate nodes [Stal85]. Acknowledgements are omitted for the sake of simplicity.

## 2.3. Layered Network Architectures

The design of computer network protocols is very complicated. To simplify this process, modern network protocols are usually designed as a hierarchy of layers. Each layer builds on the services provided by the layers beneath it, to provide a service to the layers above it. When technology changes necessitate modifications to lower layers, higher level layers can remain unchanged in a well designed system.

Except at the lowest layer, data cannot be physically transferred from a particular layer at one site to the corresponding layer at another site. Instead, entities in a particular layer (say layer *n*) will use services provided by the layer below it (layer *n-1*) to pass messages between one another. The layer *n* protocol used by peer entities in layer *n* is thus implemented by making use of the interface services provided by layer *n-1* at the boundary between layers *n* and *n-1*.

The following issues must frequently be addressed in various layers [Tane81]:

Methods for establishing and closing connections between peer entities.
Error detection and recovery.
Multiplexing of several layer *n* connections onto one layer *n-1* connection, or of one layer *n* connection onto several layer *n-1* connections.
Detection and correction of messages delivered out of sequence.
Flow control to prevent a receiver from being swamped with more data than it can handle.
Breaking long messages into manageable portions or accumulating short messages into a longer message that can be sent more efficiently.
Deciding what route a message will take.

Although layered network models have been in use for some time the protocols used in various networks, as well as the choice of layers, varied greatly from one network to another. Beginning in 1977, the International Standards Organisation (ISO) set about standardising network protocols, with a view to eventual world-wide compatibility between communications equipment supplied by different manufacturers.

ISO's "Basic Reference Model for Open Systems Interconnection" (ISO/DIS 7498, 1983), often referred to as the *OSI Reference Model*, defines a seven-layered model for computer networks. Several criteria were used in defining the layers in the model. The major considerations [Zimm81] are:

A layer should be created where a different level of abstraction is needed.
Each layer should perform a well defined function.
The layer functions should facilitate the design of internationally standardised protocols.
The layers should be chosen to minimise information flow across the boundaries.
The number of layers should be both large enough to avoid having to place distinct functions in the same layer, and small enough to prevent the architecture from becoming unwieldy.

Figure 2.5 [Hals85] illustrates the layered structure of the OSI model. The figure also illustrates the concept of *interface services* and *peer-to-peer protocols*. An interface service is a service provided by one layer to the layer above it, where both layers reside at the same physical location (for example, in the same host computer). A peer-to-peer protocol is a protocol used for communication between *peer entities* (which reside within the same layer but at different physical locations). Peer entities use the services provided by the layer beneath their own layer to implement their peer-to-peer protocol. Standardisation of the peer-to-peer protocols allows equipment from different manufacturers to be interconnected. Standardisation of the layer interface services allows the internal operation of particular layers to be modified (possibly to take advantage of new technology) without affecting other layers.

The following is a brief description [Zimm81] of some of the functions performed by the layers in the OSI model:

Application layer: Provides services to the user's application program. These services include system management, application management and distributed information handling.

Presentation layer: Manages the entry, interpretation and exchange of data. Data encryption or character code conversion would be performed by this layer. Support for virtual file and terminal access would also reside here.

Session layer: Provides support for distributed processing and the synchronisation of distributed tasks.

Transport layer: This is the lowest layer that operates on an end-to-end basis, regardless of the routing used by lower layers. It is responsible for the end-to-end transfer of data, freeing higher layers of the need for concern about this detail. Optimisation of resource usage would also be performed here; this might include upward multiplexing of several transport connections onto a single network connection (to reduce costs) or downward multiplexing of a single transport connection onto several network connections (to increase throughput).

Network layer: Responsible for the routing of data between access points. Controls the storing and forwarding of data that must pass through several stages before reaching its destination.

Data link layer: Establishes and closes data links. Performs error detection and recovery.

Physical layer: Directly controls the physical transmission medium. Performs the transfer of data bits over the medium.

Although the protocol layers used in existing networks such as ARPANET, DECNET and SNA do not correspond exactly to the layers of the OSI reference model, most of the functions found in the OSI model are present in all networks. Figure 2.6 [Tane81] illustrates one possible mapping of functions provided in the three networks named above onto the layers of the OSI reference model.

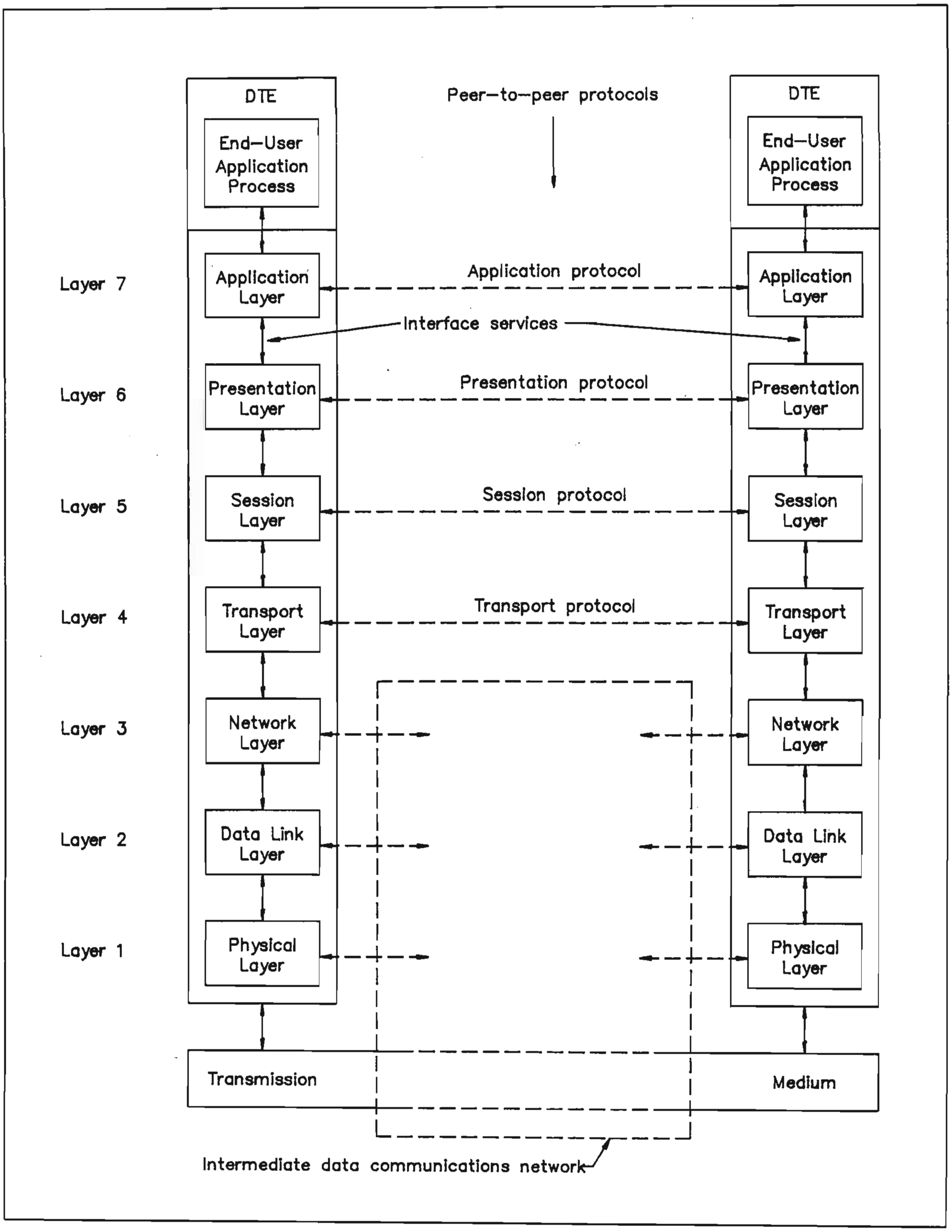

**Figure 2.5** The layered structure of the OSI reference model [Hals85].

| Layer | ISO OSI | ARPANET | SNA | DECNET |
|---|---|---|---|---|
| 7 | Application | User | End user | Application |
| 6 | Presentation | Telnet, FTP | NAU services | |
| 5 | Session | (None) | Data flow control<br>Transmission control | (None) |
| 4 | Transport | Host–Host (TCP) | | Network services |
| 3 | Network | Source IMP to destination IMP (IP) | Path control | Transport |
| 2 | Data link | IMP–IMP | Data link control | Data link control |
| 1 | Physical | Physical | Physical | Physical |

**Figure 2.6** Approximate correspondence between layers in various networks [Tane81].

## Section 3. The NUNET, EENET or CASE Network

Many of the computers (of all sizes, from PCs to mainframes) and terminals on both the Durban and Pietermaritzburg campuses of the University of Natal are connected to a terminal switching exchange network known as *NUNET* (Natal University Network). The part of the network located in the School of Electrical and Electronic Engineering (on the Durban campus) is sometimes referred to as *EENET* (Electronic Engineering Network).

The network is also often referred to as a *CASE* network, after the company that manufactures the equipment used. (Computer And Systems Engineering plc, Watford, England).

In this thesis, the term *CASE network* will usually be used to refer to any network of CASE DCX-850 nodes. The term *NUNET* will be used to refer to the University of Natal's CASE network, and the term *EENET* will be used to refer to the portion of NUNET that is located in the School of Electrical and Electronic Engineering.

### 3.1. Basic Features

Detailed information about the CASE network appears in Appendix A3. This section mentions some of the more important features.

The CASE network allows many serial (RS-232-C) lines to be interconnected in a way defined either statically, by the network supervisor, or dynamically, based on requests from the devices connected to the network.

The network consists of several *nodes*, each of which contains a CASE *DCX-850* controller [CASE-850]. Each DCX-850 node has the features of a terminal switching exchange and the nodes are interconnected in such a way that the entire network appears to the user to be a large terminal switching exchange. The DCX-850 is an extension of the DCX-840 [CASE-840], which requires an operator to configure the ports which are connected together at any moment.

Each node has several low speed RS-232-C ports, connected to computers, terminals or other devices. Nodes are interconnected by high speed composite links with an *Automatic Repeat Request* (*ARQ*) protocol, based on the *High-level Data Link Control* (*HDLC*) procedure of CCITT recommendation X.25. Low speed ports are controlled by *LSC* (*Low Speed Channel*) cards connected to the DCX-850. Low speed ports may also be connected to terminal concentrators whose composite links are connected directly to the the DCX-850 (without having to be demultiplexed).

A user at a properly configured low speed port (known as an *UMP*, or User Mapped Port) can request to be connected to any other suitably configured port on the network, subject to some restrictions that may be imposed by the network supervisor. In addition to the User Mapped Ports, the network supervisor can define ports as Internally Mapped Ports (*UMPs*) or Automatically Mapped Ports (*AMPs*). Each UMP is permanently connected to another UMP. An AMP is not permanently connected to any destination but is automatically connected to a particular destination (without any dialogue between the network and the user) when the user generates a *connect event*.

The network is designed to be used by a user at a terminal. The user will typically generate a *connect event* to get the network's attention, after which the network will establish a virtual circuit between his terminal port and a destination port. In the case of an AMP port, the network supervisor determines the destination address to which the user's terminal will be connected when a connect event occurs. In the case of an UMP port, a short dialogue between the user and the network determines the destination address. A connection between two ports is broken when a *disconnect event* is generated by either of the ports.

Low speed ports may be referred to in one of three ways: by port number, by a numeric group address, or by an alphanumeric name. The network supervisor can assign ports to groups and can assign alphanumeric names that refer to group addresses.

The network allows multiple paths to be defined between DCX-850 nodes, and is able to function in the presence of link failures. Existing connections are broken when link failures occur, but subsequent connections will use alternative routes, if they exist.

NUNET currently comprises three nodes, numbered 8, 9 and 10. The DCX-850 controller for node 8 is located in the Department of Electronic Engineering, that for node 9 is in the Computer Services Department on the Durban campus, and that for node 10 is on the Pietermaritzburg campus. NUNET is actually part of a larger, country-wide network named NIINET (National Institute for Informatics Network), administered by the CSIR (Council for Scientific and Industrial Research). Nodes 1 to 7 and 11 to 13 of NIINET are located in various parts of South Africa. The interconnections between nodes on NUNET are shown in figure 3.1. The arrangements in the School of Electrical and Electronic Engineering are shown in more detail in figure 3.2.

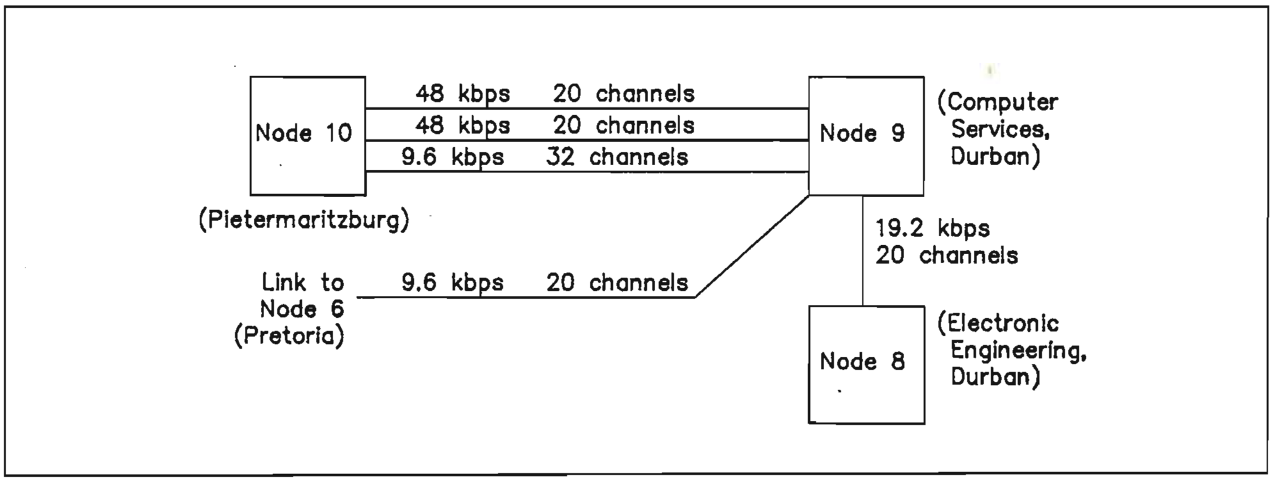

**Figure 3.1** Links between nodes in the NUNET network. The link to NIINET node 6 is also shown.

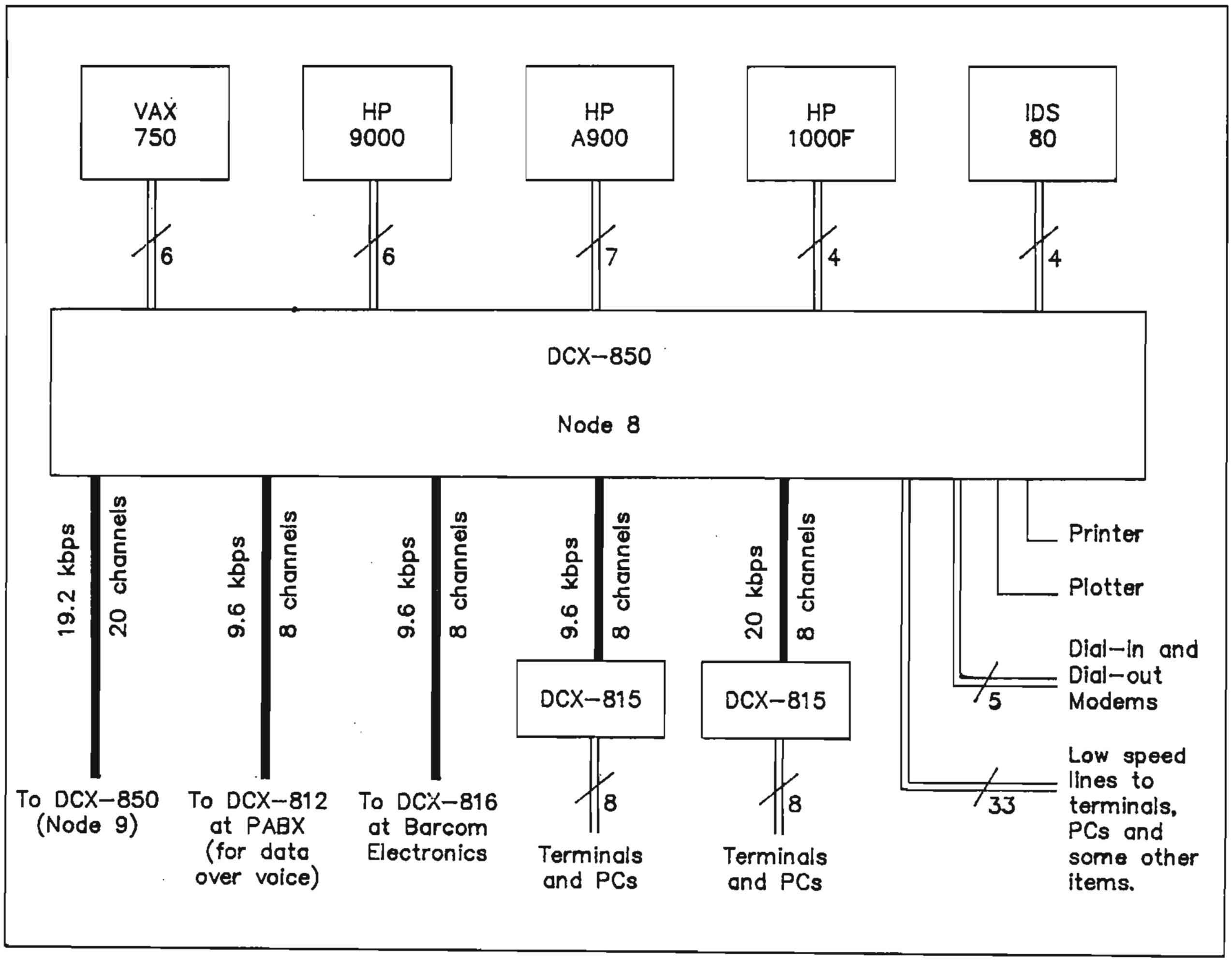

**Figure 3.2** Configuration of EENET (node 8 of NUNET).

## 3.2. The User's View of the CASE Network

Most of the ports on NUNET are UMP's, or User Mapped Ports. This means that a user whose terminal is connected to one of these ports can request the network to create a virtual circuit between his terminal and any other suitable port.

To initiate a connection, the user must generate a *connect event*. Several types of connect events are available. If a dial in modem is used, the network port to which the answering modem is connected will probably be configured in such a way that a connect event is generated as soon as the call is answered. If the user's terminal is directly connected to the network and used the RS-232-C handshake lines, the network can be configured to generate a connect event whenever the terminal is placed "on line". Where control over the RS-232-C control lines is not possible, a connect event may be generated by pressing a terminal's *break* key, or by typing a *control-T* (ASCII DC4) character.

Each port on the network must be configured by the network supervisor for the particular type of connect event that will be used. Most of the user terminal ports on NUNET use a *break* signal to generate a connect event.

In the case of an Automatically Mapped Port (AMP), the network will immediately attempt to establish a connection to the destination specified by the network supervisor as soon as a connect event occurs. A short message will indicate the success or failure of this attempt. The message COM indicates a successful connection.

In the case of a User Mapped Port (UMP), the user must type a carriage return character within ten seconds after the connect event. The network then prints a short message inviting the user to select a destination. A port address, group address or alphanumeric name may now by typed by the user. This is followed by typing a carriage return character. The network will then attempt to make the requested connection, and a short message (COM in the case of success) informs the user of the success or failure of the connection.

When a connection to a group address or alphanumeric name is attempted, the name or group may have several ports associated with it. The network will then attempt to make a connection to any available port within the group. This feature allows several network ports connected to the same computer to be grouped under a single name, so that a user can request a connection to the computer by giving the computer's name instead of having to remember the port numbers connected to the relevant computer.

If a connection attempt from an UMP is unsuccessful, the network gives the user an opportunity to select an alternative destination. At this point, the user may request the network to enter his connection request on an automatic retry queue. The network will then re-try the connection attempt at regular intervals, until a connection is successfully established. The user simply types the letter Q and a carriage return character to enable this feature. Queueing is automatically performed, without any user intervention, if a connection attempted from an AMP is unsuccessful.

Once a connection has been established, the two ports can communicate almost as though they were directly connected together. The network checks for *disconnect events*, and may perform flow control.

Flow control may use either XON and XOFF characters, or the RS-232-C control lines. The network relays flow control requests from one port to another, and may also insert its own flow control requests (to protect the buffers in the network from being overrun). Conversion between one type of flow control and another may also be done (for example, if the user's terminal uses XON/XOFF flow control and the computer to which it is connected uses hardware flow control).

Either of the connected ports can break a connection, by generating a *disconnect event*. A disconnect event can also be generated by a user to cancel a previously queued connection request that has not yet been completed. There are several possible disconnect events, and the network supervisor must configure each port to use one of the available methods. Terminals that can manipulate the RS-232-C control lines may generate a disconnect event when they are placed "off line" or in "local mode". Other ports may generate a disconnect event when a *break* signal appears on the data line, or when a *control-T* (ASCII DC4) character is transmitted. Modem connections can use a feature known as *dual level disconnect*, in which the user can send a *break* signal to close a network connection without having the telephone call terminated.

Due to buffering in the network, some data may have been sent from one port but may not yet have arrived at the destination port when a disconnect event occurs. Any such data is immediately discarded by the network.

## 3.3. Relationship to the OSI Reference Model

The CASE network does not fit very well into the layered structure of the OSI reference model. The network user does not have access to any layer services of the type used in the OSI reference model. Data cannot be sent transparently through the network, as would be required by the higher layers of the OSI model. Instead, the network user must be aware of the special effects caused by certain control characters or conditions on the RS-232-C control lines.

Within the network itself, but inaccessible to the user, features at levels up to the network layer of the OSI reference model exist. The establishment of a connection between ports on different nodes can be performed without regard to routing requirements, which is a service offered by the network layer. Error detection and recovery is performed on composite links between nodes or between terminal concentrators and network nodes; this is a function of the data link layer.

From the terminal user's point of view, the services provided by the network do not fit neatly into any layer of the OSI reference model. For example, the user accesses the network's RS-232-C port at a physical level, and the establishment of a connection implies the presence of network layer functions available to the user.

The CASE network could be used in a network conforming more closely to the OSI reference model. This would require the provision of suitable network access software in host computers connected to the network. This software would ideally be integrated into the operating systems of the host computers. Special terminal interface processors would have to be provided instead of allowing terminals direct physical access to the CASE network. The problems posed by such an arrangement are essentially the same as those faced by a network that wishes to use the public telephone system to carry its messages.

## Section 4. Enhancing the Network

This section deals with the important features of the enhancements made by the author to the CASE network in the Department of Electronic Engineering at the University of Natal. More detailed information about the implementation of the solutions described here can be found in sections 5 and 6, as well as in Appendices A5 to A12.

### 4.1. A Help Facility for the CASE Network

#### 4.1.1. Introduction

Users of a network like the CASE network often desire some type of help facility. A help facility could explain the reasons for error messages, explain how the network should be used, and provide a list of services available on the network.

A suitable alphanumeric name, such as "HELP", can easily be defined by the network supervisor, and associated with a computer that is able to provide help to network users. A user would then be able to type the word HELP just after generating a network connect event, and his terminal would be connected by the network to the computer that hosts the help facility.

The help facility could run on a dedicated microcomputer, or could use an existing port on a minicomputer. Each option has advantages and disadvantages, which are described below.

Some advantages of using a terminal port on an existing minicomputer are:

a) No new hardware would have to be designed or purchased.

b) Software development would be relatively simple; all the development tools available on the host system could be used.

c) Much disk space would be available for elaborate help facilities (although very elaborate help facilities would probably not be required).

d) If the demand proved sufficient, additional terminal ports could be assigned to the help system.

Some disadvantages of using a terminal port on an existing minicomputer are:

a) At least one of the terminal ports on the host would no longer be available for general use.

b) Some disk space on the host system would be used, leaving less free space for other purposes. Other resources on the host, such as CPU time, would also be affected. The impact on the host system's resources would be very minor, however.

c) The help system would be unavailable if the host system were taken down for maintenance, or if it failed for any reson.

d) Software written for the help system would probably not be very portable, due to the necessity of using low-level operating system functions for such things as timeout handling and single-character keyboard input.

Advantages of using a dedicated microprocessor system include the following:

a) Because the system would be dedicated to a single task, the probability of operating system crashes might be reduced.

b) A simple hardware system could readily be duplicated, for use on other CASE networks or on other nodes in the same network.

Some disadvantages of using a dedicated microprocessor system are:

a) Either hardware design and construction, or the purchase of suitable hardware, would be required.

b) A system with limited memory space, and primitive or nonexistent mass storage facilities, would make software development difficult.

c) Even if a system with floppy disks (or some similar mass storage device) were used, software development might prove difficult. (This problem would not arise if the system were purchased with an operating system, compilers and other tools, or if cross-compilers and other development tools were available on another system.)

It was decided to use the existing HP-A900 computer as the host for the help facility. The cost of a dedicated microcomputer to serve as a help provider did not seem justified, and the HP-A900 would be affected only very slightly by the loss of one of its network ports.

A simple program was written to provide the required help facility. This program presents information read from a disk file to the user one screenful at a time. Two separate disk files are used, for help and for news. The contents of these files can be changed using the HP-A900's standard text file editor. The program counts the number of lines displayed to determine when to pause, and there is also provision for pause positions to be explicitly indicated in the data files.

### 4.1.2. Using the Help Facility

Assuming that the help and news system is properly installed, a network user can use the facility by following these steps:

a) Generate a *connect event*, to get the attention of the network. If the terminal is connected directly to EENET, this is usually done by pressing the terminal's *break* key followed by typing a carriage-return, although other methods may be required, depending on the port configuration.

b) When the network issues its prompt allowing a destination to be selected, type **HELP** and press *return*. The destination name must be given in upper-case characters.

c) The network should respond with a message, which will usually be one of the following.

| | |
|---|---|
| COM | The connection to the Help facility has been made. |
| MOM | The Help facility is busy or some network resource is unavailable. The network will spend a few seconds waiting to see if a connection becomes possible, and will then respond with another message, usually COM or OCC. |
| OCC | The Help facility is busy. When this message is given, the network can be requested to wait until the facility becomes available; This is done by typing Q and a carriage-return in response to the next prompt. The network will then queue the connection request, and will respond with a COM message when the connection is eventually made. The queue request can be cancelled at any time by generating a *disconnect event*. |

Once connected to the HELP facility, typing any character will wake the program up. This is necessary because a program on the HP-A900 computer has no way of detecting when a connection to the computer has been made.

The program prints the date and time, and gives the option of obtaining Help, obtaining News, or aborting the session. The user types **H**, **N** or **A** to select the desired action.

If Help or News is selected, the program will print the relevant information, pausing at the end of each screenful of text. When the program pauses, typing an **A** will abort the session, typing an **R** will make the program restart from its first prompt, and typing any other character will make the program display the next screenful of information. It is not necessary to type a *carriage-return* character after typing a response; the program reacts immediately the relevant character is typed.

When the program reaches the end of the data file containing the information being displayed, it loops back to its first prompt, in the same way as it does when the user requests a restart by typing an **R**.

If the user takes too long to respond when the program wants input, the message **<TIMEOUT>** will be displayed and the connection will be broken. The connection is also broken if the user types an **A** (for *Abort*) in response to any of the program's prompts.

## 4.2. Peripheral Sharing

### 4.2.1. Introduction

It is often useful to allow a peripheral device to be accessed by several computers. This is especially so when the device is expensive and when its utilisation by a single computer would be fairly low. Allowing shared access to the device would allow its cost to be spread and its utilisation to be increased.

One way of allowing several computers to share a peripheral device would be to connect the device directly to a computer (in the same way as if the device were not shared at all) and to allow other computers to connect through a network to the computer controlling the device. A true computer network could make access to the shared device completely transparent to any user software. Provision of such a facility using the CASE network and the computers currently in operation in the Electronic Engineering Department at the University of Natal would be extremely difficult.

If software that produces output for a peripheral device can write its output to a disk file instead of insisting on a direct connection to the device, then file transfers could be used to transport the output file from the computer that generated the output to the computer that controls the peripheral device. This would allow a peripheral device connected directly to a single computer to be shared (in a limited sense) by other computers. Such a facility could be provided using the resources at our disposal.

If a peripheral device can use an RS-232-C serial line for communication with a computer, then it could be connected to the CASE network instead of being connected directly to any one computer. When a computer wished to use the device, a connection through the CASE network could be created, after which software running on the computer could treat the device almost as if it were connected directly to the computer. This method of peripheral sharing has been implemented and is covered in more detail in the following subsections.

### 4.2.2. Accessing Peripherals Through the CASE Network

Peripherals such as printers and plotters can easily be connected to the CASE network and given alphanumeric names. It is also possible to group several peripherals under the same network alphanumeric name, so that any available device could be used when required.

For a computer to access a peripheral device via the CASE network, several factors must be considered:

a) The computer must be able to communicate with the peripheral using an ordinary RS-232-C serial line.

b) The hardware handshake lines (CTS, DTR, DCD etc.) used in the RS-232-C interface can be used with the CASE network, but most of the computers, terminals and peripherals on EENET leave these lines unconnected. The ability to communicate using a three-wire interface (ground, transmitted data and received data) is a decided advantage in our environment, although low speed ports on the network can be configured to use the hardware handshake lines.

c) Some of the ASCII control characters cannot always be transmitted through EENET. The DC1 and DC3 (or control-Q and control-S) characters are used for XON/XOFF flow control, and the DC4 character (control-T) is used to generate connection or disconnection events. It is possible to configure low speed ports so that they do not attach any special meaning to these control characters (and use the hardware handshake lines instead), but this must be done for both ports that participate in a particular connection if it is required that all the control characters be passed transparently through the network.

d) XON/XOFF flow control is used on most low speed ports on EENET. It is therefore desirable that both the computer and the peripheral device be capable of using XON/XOFF flow control.

Before an application program can use the peripheral device, it is necessary for a connection to be made through the CASE network, between the computer and the peripheral. When the application program has finished using the peripheral, the connection should be closed.

Ideally, the network connection should be handled in a way that is transparent to the user. The user should be able to specify the name of the desired peripheral device, and the application software should be able to determine whether the device is connected directly to the computer or whether it should be accessed through the network. This facility could be built in to new software, but cannot readily be added to existing application programs.

A method of allowing users to explicitly make and close connections between their computer and a desired network peripheral is therefore needed. Once the connection has been made, the application software can use the device as if it were connected directly to the computer.

Programs have been written on four of the Department's minicomputers, allowing users to create and close connections through the CASE network. Transparent methods of accessing printers through the network have been implemented on three of these minicomputers, and a transparent method of accessing pen plotters through the network has been implemented on one of the minicomputers.

### 4.2.3. Special Adapters to Allow Binary Data Transfer

If it is absolutely necessary to transfer binary data between a computer and a peripheral device, using a network connection that does not allow certain control characters to be transmitted transparently, special adapters could be designed to facilitate this. One such adapter could then be placed in the serial link between the computer and the network, with a complementary adapter between the network and the peripheral.

These adapters could perform any desired data or protocol conversion. They would be responsible for preventing buffer overruns (in the network as well as the computer and the device) and would ensure that information seen by the network was free of the illegal control characters. Complicated features like automatic retries in the event of data corruption or transmission failure could also be built into the adapters. Another useful feature would be a watchdog timer, to break the connection after some period of inactivity.

Adapters of this type have not been implemented, so binary data transfers through the CASE network are not currently possible, except between low speed ports that have both been configured to use the RS-232-C control lines for flow control and the generation of connect and disconnect events.

## 4.3. File Transfers

### 4.3.1. File Transfers Using Terminal Emulator Programs

For simple transfers of text files, readily available terminal emulator or communications programs allow a PC user to connect through the CASE network to another computer and log-in to that computer. He can usually then instruct the remote computer to send the contents of a text file, and have his terminal emulator program capture the incoming data. Similarly, the remote computer can usually be instructed to accept data transmitted, by the terminal emulator program, from a file on the PC.

This approach is suitable only for files that contain only printable characters, and even then is not very satisfactory, because no error checking is performed, so data may be corrupted in transit between the two computers.

A user logged in to a minicomputer will often not have access to a terminal emulator program of the type found on PCs. Even if one is available, the user will have difficulty sending certain control characters to the secondary communications line used to communicate with the remote computer.

### 4.3.2. The Kermit File Transfer Protocol

Reliable file transfers require an error detecting and correcting protocol. One widely available file transfer protocol is Kermit [Kerm85], which was developed at the University of Columbia. More information about Kermit appears in Appendix A4.

Kermit is able to transfer binary or text files. The files are transmitted in packets, and each packet is acknowledged before the next packet is transmitted. Data is encoded in such a way that only printable ASCII characters are transmitted, so no control characters can interfere with the operation of the CASE network. Implementations of the Kermit file transfer protocol are available on most of the PCs and minicomputers in the Department of Electronic Engineering at the University of Natal.

PC implementations of Kermit usually include a terminal emulator capability, so using Kermit to transfer files, through the CASE network, between a PC and a minicomputer involves the following steps:

1) Run Kermit on the PC.

2) Use the PC Kermit's terminal emulator mode to connect through the CASE network to the remote computer, and log-in to the computer.

3) Run the remote computer's Kermit program.

4) Use suitable Kermit commands on the PC and the remote computer to transfer the desired files. This often requires switching in and out of the PC Kermit's terminal emulator mode, so that commands typed on the PC keyboard are interpreted by either the PC Kermit program or by the Kermit program on the remote computer, as appropriate.

5) Log out of the remote computer and ensure that the CASE network connection is broken.

6) Terminate the Kermit program on the PC.

Using Kermit for file transfers between two minicomputers is similar to the procedure just described. A possible problem is that many minicomputer Kermit implementations provide only for *remote mode* operation, in which the communications line used to communicate with the user is the same as that used for file transfers. This is the mode in which a minicomputer Kermit program operates when a PC user follows the procedure described above. File transfers between two minicomputers require at least one of the minicomputers to support *local mode* operation, in which the communications line used for file transfers is not the same as that used to communicate with the user's terminal. As a last resort, it is possible to perform the file transfer in two steps, first from the source computer to a PC and then from the PC to the destination computer. This approach requires sufficient disk space on the PC to store the file before re-transmission, and takes approximately twice as much time as a direct transfer would.

Given a Kermit program capable of local mode operation, a minicomputer user must have access to a secondary RS-232-C line, which will be connected through the CASE network to the other computer. Some way of making the connection through the network is also required. Because of the way most of the ports on EENET are configured, a user cannot simply use the terminal emulator mode provided by the minicomputer Kermit program to create a connection through the CASE network. Creating a connection requires the computer to send a *control-T* character to the secondary communications line, but if the user typed a *control-T* character at his terminal then the connection between the terminal and the local minicomputer would be broken.

A special program could be provided to allow the minicomputer to create and close connections through the CASE network. The standard minicomputer Kermit program could then use the connection for file transfers. The fundamental features of this arrangement are identical to those of the peripheral sharing arrangement described earlier, and indeed the same programs that have been provided on four of the Department's minicomputers to facilitate access to peripherals through the network can be used to make connections between minicomputers for the purpose of file transfers.

### 4.3.3. More Transparent File Transfer Methods

Using the Kermit file transfer procedure described in the previous subsection requires much effort on the part of the user. He must run a program to create a connection between his local minicomputer and the remote minicomputer, then use the Kermit program interactively to transfer files between the two computers, and he must finally run another program to close the network connection between the two minicomputers. While he is transferring files between the two computers, the user is actually logged in to both computers, which can be confusing. Depending on the state of the Kermit program on the local minicomputer, commands typed at the user's terminal might be interpreted on either the local or the remote computer.

Many Kermit implementations have a facility for using auto-dial modems; this section of a standard Kermit program could be modified to make or break connections through the CASE network instead of through the public telephone network. This would make it easy for the user to create network connections from within Kermit, instead of having to do so before starting the Kermit program.

If sufficient terminal ports were available on a minicomputer, a Kermit program could permanently be left running on it. Anyone connecting through the network to that port would then be able to do file transfers without having to log-in to that computer. Kermit programs often have a *server* mode, which allows the remote Kermit to run non-interactively. When the remote Kermit is in server mode, the user gives all his commands to the local Kermit program, and the local Kermit sends special command packets to the server to initiate file transfers. If a Kermit server were left permanently running on a port on a minicomputer, users connecting through the network to that port would be able to instruct their local Kermit program to transfer files to and from the remote computer (subject to any security provisions that might be imposed) without ever typing a command that was not interpreted by their local computer. This would greatly reduce the effort required for file transfers.

It would be easiest for a user if he could log-in to a computer and give a simple command to transfer one or more files between his computer and another, without having to worry about creating a network connection or issuing Kermit commands. Useful additions to such a program would include a way of queueing the request (so that the user would not have to wait for the transfer to complete), and a way of allowing a user on computer A, for example, to request that a file on computer B be copied to computer C.

Such a system would require a program capable of understanding the user's request, creating a link through the network to the other computer, transferring the files, and notifying the user of the success or failure of the operation. Most Kermit implementations can work non-interactively, with command line options telling them what to do, and this feature could be used by the control program on the local computer.

Having a Kermit server running permanently on each computer would reduce the difficulty of implementing such an automated file transfer procedure. Programs on the local computer would then not need to be capable of logging in to the remote computer, they would only have to connect to the appropriate port and begin issuing Kermit Server commands.

File transfers between the computers on EENET are probably not sufficiently frequent to warrant a fully automated procedure such as that contemplated above, especially if that required permanently running a Kermit server program on a dedicated serial port on each computer. The allocation of seldom-used, dedicated ports on the computers would reduce the number of ports available for general use on each computer, and some of the computers already have too few ports to cope with the peak demand.

Commands have been provided on three of the Department's minicomputers to simplify the process of using Kermit for file transfers. An automated file transfer procedure, using a standard Kermit implementation and the network connection software developed by the author, has been implemented between two of the Department's minicomputers, by A. M. Mvinjelwa [Mvin87].

## Section 5. Programs to Create Network Connections

### 5.1. Introduction

In order to support peripheral sharing and file transfer through the CASE network, a program capable of creating and closing network connections is required. The interaction between this program and the network would be the same as that between a terminal user and the network.

Before a network connection can be established, the program must have access to an outgoing RS-232-C line between the computer and the network. It must then generate a *connect event*, and must respond in the same way as a user would to the messages printed by the network during the connection dialogue. This includes the ability to detect error conditions, and either to recover from the errors or to cease trying to make the connection.

The connection may be made successfully, or may be unsuccessful. In either case, the connection program needs some way of returning status information to the user or program that invoked it.

After the connection has been used for the desired purpose, which is likely to be either peripheral sharing or file transfer, the connection needs to be closed. This is done simply by generating a *disconnect event* on the serial line used to access the network. The outgoing line should then be de-allocated, so that it can be used by other programs or users.

It may be seen that there are four major requirements for a program used to manage CASE network connections. The program must determine what the user wishes to do (create or close a connection); it must secure exclusive access to an outgoing RS-232-C line, so that there is no interference with other users; it must interact with the network in order to create or close a connection; and it must report its success or failure in some way. Another useful feature would be the maintenance of an event log, to aid in software debugging, statistical analysis, or accounting.

Implementing a program to create and close CASE network connections is greatly simplified if the computer system provides input "type ahead" buffering. This is useful because, during the connection dialogue, the network outputs several lines of information very rapidly, and the program may not be able to process the information as fast as it is sent by the network.

Programs to create and close network connections have been written for four of the Department of Electronic Engineering's five minicomputers. Almost identical versions of a program known as the **Connect** program, written in Fortran-77, run on three of the minicomputers: a VAX-750 (with the VMS operating system), an HP-A900 (RTE-A operating system) and an HP-9000 (HP-UX operating system). A different program, written in Fortran-IV, runs on the Gerber IDS-80 computer (DMS-2 operating system).

The fifth minicomputer, an HP-1000F (RTE-6 operating system) cannot readily cope with high speed input from a terminal, so messages sent by the network controller are not received properly. The Connect program relies on some input type-ahead buffering by the operating system, to avoid having input from the network controller lost or garbled. Special software could possibly be written to cope with the high speed input, but standard RTE-6 methods (including multiple-buffered *CLASS I/O*) were found to be inadequate. The HP-1000F computer is due to be scrapped in the near future and work previously performed by it is being transferred to other machines, so it was decided not to support the Connect program on the HP-1000F.

## 5.2. The Connect Program

The program used on the HP-A900, HP-9000 and VAX-750 computers to create and close connections through the CASE network is known as the **Connect** program. This program's interaction with the network is controlled by command files written in a simple command language. Standard command files have been written to allow the program to create and close CASE network connections, but the program itself is sufficiently general that it could be used for other purposes.

The following subsections discuss various aspects of the Connect program. More detailed information appears in Appendix A7.

### 5.2.1. System Dependencies

The Connect program needed to use system-dependent subroutines for input and output to the CASE port, because standard Fortran READ and WRITE statements cannot perform character-oriented I/O or timeout handling. In addition to the obviously system dependent tasks of communicating with the network port, some other system dependencies proved necessary.

When writing to the user's terminal, some systems require carriage-control characters and others do not. Different systems have different, non-standard ways of specifying that a disk file should be opened for reading only, or that simultaneous access by other programs is allowed. Although these and other differences are minor, and a generic program could possibly be written to work on all our computers, a method of conditional compilation appeared desirable.

The assembly languages on many computers allow a program source file to contain lines that may or may not be assembled, based on the values of expressions evaluated at assembly time. Few common high level languages have similar features (*C* is an exception). To make it easier to maintain slightly different versions of the same program on several computers, a fairly simple conditional preprocessor, called Condpp, was written. This program is described in Appendix A6.

Apart from some highly system dependent modules, the master source code for the Connect program contains statements that may or may not be needed on a particular target computer. Before compiling a version of the Connect program for a particular computer, the Condpp preprocessor is used to extract a suitable source code version from the master source code. Modifications to the master source code can easily be distributed, without the need for any manual editing on the target system.

### 5.2.2. Command Files

The primary function of the Connect program is to interpret specially prepared command files. The command files are initially prepared in a printable form, but may be converted to a binary form by the ConnToBin program. The Connect program can interpret either printable or binary command files, but binary command files are interpreted much more efficiently that are printable command files.

Each line in a (printable) command file used by the Connect program contains either a command or a comment. Blank lines and lines whose first non-blank character is a star (*), semicolon ( ; ) or exclamation mark ( ! ) are comment lines. Other lines contain commands. A comment may be appended to a command line if it is preceded by a semicolon ( ; ) or exclamation mark ( ! ). Commands may not cross line boundaries, and no more than one command may appear on a line.

Tokens that may appear in the command file include un-quoted words and quoted character strings. Character strings may be enclosed in either single or double quotation marks ('...' or "..."). An un-quoted word may be used wherever a quoted character string is allowed, but the converse does not apply. Within a quoted character string, a quotation mark of the type used to delimit the string may appear twice in succession; this has the effect of inserting a single such quotation mark into the string, as in many computer languages. A special syntax may be used to specify non-printable characters in any character string.

Any line may begin with a label. A label is an un-quoted word whose last character is a colon ( : ). The label may be used as the destination of a conditional or unconditional **GOTO** command. A line may have any number of labels, and a line with labels need not also include a command.

Wherever a character string is permitted, a symbol reference may be used. A symbol reference is an un-quoted word whose first character is a dollar sign ($). The name that appears after the dollar sign is the name of a symbol whose value is to be used. Symbols may be assigned values in the command line used to run the Connect program, or by a **SET** command that appears in the command file. Symbol names are not case-sensitive. Some symbols are automatically set by the program in various circumstances; these symbols may be referred to freely in the command file, in exactly the same way as other symbols are used.

Each command line specifies a simple operation that must be performed. Operations that may be performed include defining the network connection device (**DEV**), setting the maximum step count (**MAXSTEPS**), sending data to the network connection (**SEND**), sending data to the user's terminal (**SAY** and **SHOW**), receiving data from the network connection (**RECEIVE**), setting symbol values (**SET**), conditional and unconditional transfer of control (**RECEIVE**, **IF** and **GOTO**) and program termination (**ABORT** and **FINISHED**). The conditional and unconditional **GOTO** commands are the only method of flow control available in a command file.

Whenever a command is executed, a step counter is incremented. When the step count exceeds a threshold determined by the **MAXSTEPS** command (or by the **MAXSTEPS** symbol, if it was set in the command line used to run the Connect program), the program will abort.

## 5.2.3. Command Line Invocation

All the computers on which the Connect program has been implemented provide some way in which command line arguments can be passed to a program. The command line arguments passed to the Connect program name the command file to be interpreted by the program, and may optionally assign values to various symbols which will be accessible to commands in the command file.

The command line used to run the Connect program has the following format:

CONNECT *commandfile* [*symbol=value*] ...

The *commandfile* argument specifies the name of the command file to be interpreted by the program.

The square brackets [] and the ellipsis (...) above are not part of the command line, they simply indicate that the *symbol=value* part is optional and may be repeated any number of times. *Symbol* is the name of a variable that will be added to the program's symbol table, and *value* specifies the value to be assigned to the variable. The same special syntax that may be used to specify character strings inside command files may be used to specify non-printable or other special characters in the *value* string.

Arguments are separated by space or comma characters. If the required *value* specification contains space, comma, quotation mark or other special characters, the entire *symbol=value* string must be enclosed in single or double quotation marks ('...' or "..."). For example, if a user wishes to assign the value "**b c=d**" to the symbol named "**a**", he might gives the command

```
connect cmdfile "a=b c=d".
```

On some computers, quotation marks and some other characters may be used for special purposes by the command line interpreter, and may therefore not be present in the command line that is made available to a program. Special steps may need to be taken to ensure that the Connect program can see the quotation marks referred to above; any such action would of course be system dependent.

### 5.2.4. Selecting an Outgoing Communications Line

The name of the outgoing RS-232-C line to be used by the Connect program may be specified in one of two ways. A symbol called **DEV** can be defined in the command line used to run the Connect program, or one or more **DEV** commands may appear at the start of the command file.

If the DEV symbol is defined in the command line used to run the program, then the program attempts to use the specified device, and ignores any DEV lines that may be present in the command file interpreted by the program. If the DEV symbol is not defined in the command line, then the program expects the command file to begin with one or more DEV lines. The device specified in the first DEV line is the first to be tried. If it cannot be used, devices specified in subsequent DEV lines are tried. If the program cannot open any of the relevant devices, it terminates with an error message.

The device name given in the DEV symbol in the program's command line or in a DEV line in a command file is not necessarily the same as the name used by the operating system to refer to that device. This is because translation of device name aliases may be performed when the device is actually opened.

### 5.2.5. Device Locking

Because several users, or several programs controlled by a single user, might simultaneously attempt to use the same device to access the network, some method of securing exclusive access to a device is required. Many computer operating systems allow exclusive access to a device to be granted to a program, to a process (which may contain several programs), to a job or session (which may contain several processes), or to some other entity.

The VMS operating system on the VAX-750 computer allows a device to be allocated to a process; programs and sub-processes running within that process then have permission to use the device. The RTE-A operating system on the HP-A900 computer allows a program to lock a device; other programs (even if they are children of the locking program) may not access the locked device unless they use special system calls and a 16-bit key that is made available to the locking program. The HP-UX operating system on the HP-9000 computer does not support device locking (but access may be restricted to programs owned by a specific user).

Because of the large differences between various operating systems' treatment of true device locking, the Connect program uses a different method to decide whether it is permitted to use a given device. Implementations of the program on some systems may also use operating system locks, but the basic method used by the Connect program relies on cooperation between programs attempting to use a given device.

A system of lock files is used for this purpose. There is a *Device Lock File* for each device that may be used by the Connect program; the existence of one of these lock files usually means that the program that created the lock file is permitted to use the associated device. Another lock file, called the *Exclusion File*, gives the program that created it permission to create, modify or delete device lock files.

It is sometimes useful for several programs in sequence to make use of the same device. For example, the Connect program could be used to create a connection through the CASE network, then another program could use the network connection, and finally the Connect program could be used to close the connection. The Device Lock File contains a short text description of the reason for the presence of the lock, and this may be used to assist in the coordination of access to a device by several programs in sequence.

When a device is used by several programs in sequence, the first program should arrange to leave a mutually agreed code as the *lock information* in the device lock file. The second program in the sequence can verify that this special code is in fact present before it uses the device. This process can be continued as long as is necessary, with the last program in the sequence finally deleting the lock file.

When the Connect program is invoked, values may be assigned to the **OLDLOCK** and **KEEPLOCK** symbols. If the **OLDLOCK** symbol is defined, the program will refuse to use the device unless the existing lock information associated with the device is correct. If the **KEEPLOCK** symbol is defined, the program will ensure that the specified lock information will be retained when the program terminates.

The methods used to maintain these lock files are described in detail in Appendix A7.

### 5.2.6. **Error Handling and Event Logging**

Several types of error conditions may be encountered by the Connect program. Errors dealt with by the program itself include incorrect command line arguments, incorrect command file syntax and inability to secure access to an outgoing communications line. The writer of the command files interpreted by the program may allow for detection and recovery from other error conditions, such as nonexistent or busy network destinations. **FINISHED** and **ABORT** commands are used in command files to signify successful or unsuccessful termination. A final type of error occurs when the the program executes a predetermined number of commands from its command file without encountering a **FINISHED** or **ABORT** command.

When the program is successful (and executes a **FINISHED** command in a command file), it terminates without printing any special messages. If it is unsuccessful, it prints a message indicating the program's version number and revision date, and the nature of the problem. If the problem has to do with a command file, the current position within the file is also reported.

Most operating systems provide a way for a program to return an exit status code when it terminates. This status code would then be available to the parent program or command file that invoked the newly terminated Connect program. The Connect program will exit with one of three possible status codes, signifying

a) success,
b) failure due to executing too many steps in the command file,
c) failure due to executing an **ABORT** command in the command file, or due to some other major problem.

The Connect program maintains a log file, to which it appends an entry when it begins execution and when it terminates. The information written to the log file includes the date and time, user name and user terminal identification. The information written at the start of the program also includes the command line arguments passed to the program, and the information logged at the end of the program includes an indication of the program's success or failure. The information in the log file can be used as a software debugging aid, or for statistical or accounting purposes. If the log file does not already exist, the program does not create it; this allows the overhead associated with maintaining the log file to be avoided.

## 5.3. The CASE Program on the IDS-80

The Gerber IDS-80 computer does not have a Fortran-77 compiler, so the Connect program used on other computers could not be compiled on the IDS-80. The Connect program makes heavy use of *CHARACTER* variables, so could not easily be converted for the Fortran-IV compiler on the IDS-80.

The major use of the Connect program, at present, is in creating or closing connections through the CASE network. A program named CASE was written for the IDS-80, to perform these tasks. Unlike the Connect program, the CASE program does not use flexible command files. The operations required in creating or closing a network connection are hard coded into the CASE program.

The CASE program may be invoked directly by a user to create or close a connection through the CASE network, or may be invoked by another program, using standard program scheduling operations provided by the operating system.

### 5.3.1. Command Line Invocation

To create a connection through the CASE network, the CASE program may be used as follows:

`RUN,CASE,`*syslu*`,`*destination*.

*Syslu* is the system logical unit number (*LU*) of the serial port that must be used for the connection, *destination* is the name of the network destination to which the port must be connected. An optional third argument, the *debuglevel* may also be used. This third argument is an integer used to control the display of information intended for debugging. If the *debuglevel* parameter is not given, or if it is specified as zero, no debugging information will be displayed. Values 1 and 2 specify different degrees of debugging information. The *debuglevel* parameter should not be used under normal circumstances.

To close a connection, the following command is used:

```
RUN,CASE,syslu.
```

### 5.3.2. Access to the Outgoing Communications Line

On the IDS-80, the operating system refers to devices by numbers, called LU numbers. The system administrator can define a mapping between user LU numbers and system LU numbers. Each authorised user can have a different LU mapping table. The system LU number of the device (or communications line) that will be used for network access is passed as a command line argument to the CASE program.

The program assumes that the device used for the network connection is one of the four ports on a version 2 RMUX card [Pepl88]. The RMUX card and the associated driver software allows input type-ahead buffering and XON/XOFF flow control, which the CASE program relies on to avoid data loss.

The program makes no attempt to lock the LU that it uses for the network connection. Other programs that schedule the CASE program may lock the device (using the system LURQS subroutine) before invoking the CASE program. This method of locking, if it is done by the program that invokes the CASE program, does not prevent the CASE program from being accessing the LU but does prevent access by other users.

### 5.3.3. Error Handling

If at first the program cannot successfully open a CASE network connection, it tries again. It will stop trying after a predetermined number of operations. This action is very similar to that of the Connect program described in section 5.2, when the Connect program is used with a command file designed to create a connection through the CASE network.

When the program is successful, it terminates with an exit status of zero, and when it is unsuccessful it displays an error message and terminates with an exit status code of 1. If the CASE program was scheduled by another program, the exit status code is available to the other program.

## Section 6. Building on the Connection Programs

The following subsections describe several functions which have been built on top of the basic connection programs described in section 5. Topics covered include standard command files used with the Connect program, user commands that simplify the use of the Connect program, the integration of peripheral sharing facilities into existing software, and file transfers through the CASE network.

### 6.1. Standard Command Files for the Connect Program

Identical versions of some standard command files for the Connect program are available on all three computers that support the program. These command files are described in the following sub-sections, and all except **UNLOCK** are listed in Appendix A8.

#### 6.1.1. The TOCASE and CASEQ Command Files

The **TOCASE** and **CASEQ** command files are both used for creating connections through the CASE network. Each expects a symbol named **DEST** to contain a CASE network destination address, and will attempt to create a connection through the network to that destination. (The **DEST** symbol must be defined in the command line used to run the Connect program.)

The TOCASE command file is usually used when a user at a terminal will be able to retry a connection attempt manually, if it fails. The CASEQ command file is designed for non-interactive use.

The TOCASE command file frequently prints messages on the user's terminal to report on the progress made. It also prints all data received from the network (after encoding control characters in a printable way). The CASEQ command file, on the other hand, does not print anything on the user's terminal.

These command files require the word **SELECT** to be present (in upper- or lower-case) on the last line of the prompt message printed by the network when a connect event occurs. If the word **SELECT** does not appear, then the command files will not know when to output the destination address.

Both command files recognise several of the error messages that may be reported by the CASE network, and abort if one of these error message is encountered. The **OCC** message, which indicates that the device at the destination address is busy and therefore unavailable, is treated differently by the two command files, however.

The TOCASE command file treats the OCC message in the same way as any other error, and aborts.

When the CASEQ command file encounters the OCC message, it queues the connect request. This is done by responding with the character Q instead of a destination address when the next network prompt is received. If the queue attempt is unsuccessful, the command file aborts. If the attempt is successful, the command file waits for the connection to be made, which will be signaled by a COM message from the network.

Both the TOCASE and the CASEQ command files will retry the connection attempt until either it is successful or they recognise an error message which causes them to abort, or until the maximum step count specified by the **MAXSTEPS** command is exceeded.

If the program aborts (and especially if the abort is due to too many command file steps being executed), it is possible that a connection has actually been made, but has not been recognised properly. To avoid the possibility of tying up the network resources indefinitely, a failed connection attempt should be followed either by an explicit disconnection or by another connection attempt. The **FROMCASE** command file may be used for the disconnection.

The following examples demonstrate how these command files can be used. The network destination address is not interpreted in any way by these command files; it is simply passed on the the network.

```
connect tocase dev=57 dest=8.43
```

Use device **57** to connect to CASE network address **8.43**.

```
connect caseq dev=lp dest=ANADEX keeplock=lp_123
```

Use device **lp** to connect to the network destination named **ANADEX**. If the connection is successful, do not unlock the device, and set the *Lock Information* in the *Device Lock File* to "**lp_123**".

### 6.1.2. The FROMCASE Command File

The FROMCASE command file is used to close a CASE network connection. A symbol named **DELAY** may optionally be defined in the command line used to run the Connect program. If this symbol is defined, then its value is a number of seconds which should elapse before the connection is closed. The default delay, if the **DELAY** symbol is not defined, is 1 second.

The following examples illustrate the usage of this command file:

```
connect fromcase dev=57
```

Close the CASE network connection on device **57**.

```
connect fromcase delay=30 dev=lp oldlock=lp_123
```

If device **lp** is locked, and the *Lock Information* is "**lp_123**" then delay 30 seconds and close the CASE connection. A command like this might be used to close the connection created with the example given at the end of section 6.1.1.

### 6.1.3. The UNLOCK Command File

The **UNLOCK** command file contains only a **FINISHED** command. It is particularly useful when a computer is booted up, to remove obsolete lock files that may be left over from before the computer went down. For example, the following command is executed whenever the HP-A900 computer is booted up:

```
CONNECT UNLOCK DEV=57 OLDLOCK=?
```

Assigning a question mark (?) to the **OLDLOCK** symbol instructs the Connect program to delete any lock file that might already have been present for device 57.

It would be equally simply to delete the relevant lock files directly, instead of using the above method. Apart from speed (deleting the lock file is much faster than the above method), the only difference between the two methods is that the **OLDLOCK=?** approach will not remove a *strong lock* (that is, a Device Lock File whose *Lock Information* begins with an exclamation mark character). Another possible advantage of the above method is that only the names of the devices, not the names of the lock files, need to be known when the command is issued.

## 6.2. User Commands to Simplify Making Connections

To make it easier for users to create and close connections through the CASE network, commands named **CONN** and **DISC** have been provided on the three computers that support the Connect program. These commands are implemented in the command language of the standard command interpreter on the various computers. (On the VAX-750, the CONN command is spelled **KONN**, to avoid a conflict with the standard DCL CONNECT command.)

The CONN command is used to make a connection through the CASE network, and the DISC command is used to close such a connection. These commands should be used in the following way:

CONN *device destination*

```
DISC device
```

*Device* is the name of the device port on the local computer that will be used for the network connection or disconnection. *Destination* is the CASE network address of the destination port.

If the CONN and DISC commands are used without the correct arguments being given, they prompt the user to enter the appropriate information.

The CONN command uses the Connect program with the *TOCASE* command file to create a link through the network from the local *device* to the network *destination*. Other programs may then use the local device, and will have direct access to the device at the network destination. The DISC command is used to close the connection when it is no longer required.

On the HP-A900 and the VAX-750 computers, the CONN command saves the *device* name in such a way that the DISC command may later be used without specifying the *device* name. The name is saved as a command interpreter symbol.

Before attempting to run the Connect program to create a connection through the CASE network, the CONN command constructs a character string that will identify the user. This character string is assigned to the KEEPLOCK symbol when the Connect program is executed. For example, on the HP-9000 computer, if a user named **fred** gives the command

```
conn tty07 8.43
```

then the CONN command will run the Connect program as follows:

```
connect tocase dev=tty07 dest=8.43 keeplock=conn_fred
```

This instructs the Connect program to use the Connect command file named **tocase**, which creates a connection through the CASE network. The local device named **tty07** will be connected to the network address known as **8.43**. After the connection has been made (if it is successful) the Connect program will store the character string "**conn_fred**" into the device lock file, instead of removing the lock. Because the lock file is not deleted, other instances of the Connect program, possibly executed by other users, will refuse to use the device, unless the OLDLOCK option is used.

Like the CONN command, the DISC command also constructs a character string that identifies the user. This character string is assigned to the OLDLOCK symbol when the Connect program is executed by the DISC command. This arrangement allows a user to use the CONN command to create a connection, then to use some other program, and finally to use the DISC command to close the connection. While the other program is in use, the device will be protected from other users running the Connect program.

Following the example CONN command given above, the command

```
  disc tty07
```

will run the Connect program in the following way:

```
  connect fromcase dev=tty07 oldlock=conn_fred
```

If the CONN and DISC commands are used correctly, the OLDLOCK information given here will match the KEEPLOCK information stored when the CONN command was used, and the Connect program will close the network connection.

## 6.3. Peripheral Sharing

Transparent access to printers shared through the CASE network has been provided on three of the Department of Electronic Engineering's minicomputers, the HP-9000, HP-A900 and Gerber IDS-80. On these computers, users can submit print jobs in exactly the same way for printers that are accessed through the CASE network as for printers that are connected directly to the computer. No method for transparently accessing printers through the network has been provided on the VAX-750.

Transparent access to plotters connected through the CASE network has been provided on the Gerber IDS-80. Transparent plotter access methods have not been provided on any other computers, but a simple user command provided on the HP-9000, HP-A900 and VAX-750 computers allows data files to be sent through the network to a shared plotter.

The following subsections describe these arrangements in more detail.

Where peripheral sharing has not been made transparent, users can of course manually use the appropriate commands to create and close connections to the devices they wish to access.

### 6.3.1. Peripheral Sharing on the Gerber IDS-80

On the IDS-80 system, each system LU has a device type number and various type-dependent parameters. There is also a short text description associated with each system LU [IDS-SRM1]. There is no common convention regarding the information that is placed in the text description. Gerber-supplied software that wishes to use a printer or plotter, for example, often checks the device type parameters to verify that the relevant device is of the required type.

If it is desired to use the same LU as both a printer and a plotter, at different times, by swapping cables between various devices connected to the same LU on the computer, or by using a network such as the CASE network (which amounts to the same thing as swapping cables, from the point of view of the software that wishes to communicate with a printer or a plotter), then the device type database must be modified whenever the function of the LU is changed. Modifying this database is a tedious manual process, but can be done without shutting down the system. It would also be possible to perform the modifications programmatically, but no program that does so has been written.

It was desired to have access through the CASE network to one plotter and one printer. At any one time, the LU that is configured as a printer would either be connected through the CASE network to the required printer, or would not be connected through the network at all. When it was connected, it would always be to the same network destination. A similar arrangement would apply to the LU that is configured as a plotter.

Because each of the LU's just mentioned is associated with its own particular destination address on the CASE network, it was decided to use a special convention in the text field of the device information database to identify the CASE destination (if any) corresponding to a particular LU. This convention states that the text description will begin with the two characters "C=" if the device should be connected through the CASE network. The network destination is listed just after the "C=" characters. A semicolon ( ; ) terminates the network destination name. Arbitrary other information may appear after the semicolon.

A subroutine called CSCHK was written to check for the special "C=dest;" information, and to return the network destination if the special information is found. This subroutine can be used, for example, by a program that wishes to use a printer but does not know whether or not it must be accessed via the CASE network. If CSCHK reports that the device does not need to be accessed through the network, then the program can use the device directly. If, on the other hand, CSCHK reports that the device must be accessed through the network then the program should invoke the CASE program to establish a connection through the network to the device, and again to close the connection when the program has finished using the device.

Programs that use this method will work correctly whether or not the device they wish to use is connected via the CASE network. The network address corresponding to the system LU number can be changed simply by modifying the standard system device information database, and such changes will be transparent to programs that use the device.

### 6.3.1.1. **Printing from the Gerber IDS-80**

Submitting a print job on the IDS-80 involves creating a file that contains the text to be printed, and using the **SPOOL** command to add the job to a queue [IDS-SRM1]. The actual printing is done by a program called **SPMON**, which is started when the computer is booted up.

The SPOOL command and the SPMON program are both standard system components, but the SPMON program (for which the source code is available) was modified. The modified SPMON program now determines whether the output device for a particular print job must be accessed through the CASE network, and uses the CASE program to create and close the network connection, if necessary.

The changes to the SPMON program are almost completely transparent to the user. The system administrator must ensure that the device description text for the relevant LU conforms to the format expected by the CSCHK subroutine, which is used to determine whether the device must be accessed through the CASE network. The CASE program will print error messages on the system console if it fails to create the required connection. These error messages can usually be ignored, because the connection attempt will be retried until it is successful.

#### 6.3.1.2. Plotting from the Gerber IDS-80

Several standard programs on the IDS-80 produce output that must be plotted. The output is written to a file in a device-independent format. The DMS **DRAFT** command is then used to plot the data to a suitable plotter [IDS-SRM1].

There are two plotters that are generally used from the IDS-80. The first is a photo-plotter, which is connected directly to the IDS-80, and cannot be used by any other computer. The second is an HP-7586 pen plotter, which was once connected directly to the IDS-80 but is now accessed through the CASE network.

When a DRAFT command that refers to the HP-7586 plotter is used, a driver program named **PLTHP** is scheduled. The source code for this driver program is available, and has been modified so that it can use a plotter connected through the CASE network.

The new version of the PLTHP program uses the CSCHK subroutine described above to determine whether the plotter is connected directly to the computer or whether it must be connected via the CASE network. If the plotter is connected via the CASE network, the PLTHP program uses the CASE program to make a connection through the network. Once the connection has been established, plotting proceeds in the normal way.

It is possible that the CASE program will print error messages on the user's terminal, if it cannot connect to the plotter. These error messages can usually be ignored, because the connection attempt will be retried until it is successful.

### 6.3.2. Printing from the HP-A900

The standard print spooler program on the HP-A900 computer is called **PRINT** [RTE-Uti]. No source code was available for this program, so modifying it to be capable of using printers that are accessed through the

CASE network was not possible.

Another print spooler program, named **LPR**, was available on the HP-1000F computer. This program was written by D. Abbot, in 1986.

Because the source code for **LPR**, as well as its associated programs **PRNT** and **GETPRLU**, was available, it was decided to modify these programs to allow them to use printers accessed through the CASE network.

These programs have been modified both to operate correctly on the HP-A900 computer, whose operating system (RTE-A) differs slightly from that on the HP-1000F (RTE-6), and to allow completely transparent access to printers connected through the CASE network. The user specifies the required printer by name, and the program determines how the printer should be accessed.

### 6.3.3. **Printing from the HP-9000**

The standard print spooling system on the HP-9000 computer is called **lpr**, or just **lp** (both names refer to the same program). The lpr system includes many programs, most of which perform administrative functions.

A user submits a print job by running the **lpr** (or **lp**) program. The information to be printed can form lpr's standard input, or can reside in files whose names are passed as command line arguments to lpr. Optional command line arguments passed to the lpr program can be used to specify the printer or class of printer that must be used, a heading or title for the print job and various other details. Device dependent information may also be specified. This information is passed to the interface program for the particular printer used, and is not interpreted in any other way.

A print job may be placed in a queue that belongs to a particular printer; it will then be printed when that printer is available. Alternatively, a print job may be placed in a queue that belongs to a *class* of printers; it will then be printed when any printer in the class is available. The default queue, which is set by the lpadmin program, may be either a printer queue or a class queue.

Each printer may have its own interface program. For each print job, the *lpsched* program schedules the appropriate interface program, passing it arguments that describe the information to be printed. The printer interface programs are typically *shell scripts* (command files interpreted by the standard command interpreter, or *shell*), so they are easily modified.

To enable the print spooler to use printers connected through the CASE network, a new interface program was written. The new interface program is based on the interface program named **dumb**, which is for a simple line printer.

A printer named **anadex** was previously the default printer. The definition of this printer was not changed. A new printer named **case** was defined (using the lpadmin program). This printer uses the modified interface program mentioned above. A new printer class named **lp** was defined, the printers **case** and **anadex** were made members of this class, and this class was made the default print queue.

Print jobs that are sent to the default queue will now be printed on either the **anadex** or the **case** printer. Usually, the **anadex** printer is not physically connected to the system, and is marked as unavailable. This means that all print jobs are printed on the printer named **case.**

The **case** interface program uses the Connect program with the *caseq* command file to make a connection through the CASE network to the required printer. It loops until the connection is made successfully. After an unsuccessful connection attempt, the interface program delays for a short period before retrying. The length of the delay starts at ten seconds, and progressively increases as more failures occur, to a maximum of two minutes.

Once the connection has been made, the interface program may send a special character sequence to set the printer to a particular mode, according to the options specified by the user. It then proceeds to print the required job, in the same way that a standard interface program would do, and finally uses the Connect program to close the connection.

### 6.3.4. Printing from the VAX-750

There is no automated method for submitting print jobs for a printer connected to the VAX-750 computer through the CASE network. The standard PRINT command works only with a printer that is directly connected to the computer.

The VMS operating system allows various queues, including print queues, to be defined, using the DCL command **INITIALISE/QUEUE.** Each queue has associated with it a program called a *symbiont,* which interacts with both the system queue manager and the input or output devices associated with the queue. A symbiont may control up to sixteen queues, all of which must be of the same type.

When a print queue is defined, a user-defined *symbiont* program may be specified [VAX-DCL]. This symbiont will then be used instead of the standard print symbiont, **PRTSMB.** A user-defined print symbiont could be written to handle connection through the CASE network to a printer. This would be a difficult task and has not been done.

### 6.3.5. The CPLOT Command

If a program that produces output for a device is able to write its output to disk instead of insisting that the device be directly connected to the computer, then the CPLOT command can be used to facilitate access to that device through the CASE network.

The function of the CPLOT command is to create a connection through the CASE network to some destination, send the contents of a data file to that network destination, and close the connection. The most common use of this command is to send plot files generated by some program to a plotter connected to the CASE network, and the command's name is derived from this usage.

The command is used as follows:

CPLOT *device destination datafile*

The *device* and *destination* arguments have the same meaning as in the CONN command. *Device* is the name of the communications line on the local computer that will be used for network access. *Destination* is the CASE network destination address. The *datafile* argument is simply the name of the data file that must be sent to the network destination.

The data file name is optional; if it is omitted, the file **HPPLOTTER.DAT** will be used. A data file with this name is commonly produced by a graph drawing program named GRAPH, which is available on all three computers that support the Connect program [Barr86].

The CPLOT command is implemented on each computer using the control language of that computer's standard command interpreter. If the CPLOT command is used without the correct arguments, it prompts the user to enter suitable values.

The following examples illustrate the use of the CPLOT command:

```
cplot txa5 7586
```

Use device **txa5** to connect to the network address known as **7586**. Then send the file **HPPLOTTER.DAT** to the device. Finally, disconnect.

```
cplot txa5 7586 other.dat
```

As above, but use the data file **OTHER.DAT** instead of the default **HPPLOTTER.DAT.**

### 6.4. File Transfers

Transferring files between minicomputers connected to the CASE network is done using the Kermit file transfer protocol [Kerm84] [Kerm85]. Implementations of Kermit are readily available for many computers, including the Department of Electronic Engineering's HP-9000, HP-A900, VAX-750 and IDS-80 computers.

Kermit programs may be used interactively, in conjunction with programs that create and close connections through the CASE network, to transfer files between minicomputers connected to the network. Automated file transfer procedures can be provided, allowing the details of making connections and using Kermit to be hidden from the user. The following subsections address these issues.

#### 6.4.1. Using Kermit for File Transfers Between Minicomputers

Kermit programs on personal computers (PCs) are usually interactive in nature, and support *local mode* operation, in which the communications line used for file transfers is not the same as the line used to connect the user's screen and keyboard to the computer. Kermit programs on minicomputers usually support *remote mode* operation, in which files are transferred using the same communications line that is used to communicate with the user's screen and keyboard. Some Kermit implementations, including those on the HP-9000, HP-A900 and VAX-750 computers, support both remote and local mode operation.

A user logged in to a minicomputer that has a Kermit implementation that supports local mode operation and has a *terminal emulator* mode that allows commands typed by the user to be relayed to a remote computer, can use the following procedure to transfer files between that minicomputer and another:

1) Create a connection through the CASE network to the remote computer.

2) Use the local Kermit program's terminal emulator mode to log-in to the remote computer and run its Kermit program.

3) Use the two Kermit programs to perform any desired file transfers, switching the local Kermit program into and out of its terminal emulator mode whenever necessary.

4) Terminate the remote Kermit program and logout from the remote computer.

5) Terminate the local Kermit program.

6) Close the CASE network connection.

### 6.4.2. The KERNET Command

The KERNET command, which is provided on the HP-9000, HP-A900 and VAX-750 computers, simplifies procedure described in the previous subsection. The command is used as follows:

KERNET *device destination*

The *device* and *destination* arguments have the same meaning as in the CONN command. *Device* is the name of the communications line on the local computer that will be used for network access. *Destination* is the CASE network address of the remote computer.

KERNET uses the CONN command to make a connection through the CASE network to the required destination. It then invokes the Kermit program in such a way that Kermit knows which local device must be used for the remote connection. The user then enters into an interactive dialogue with the Kermit program, using it to perform file transfers or other operations. Finally, KERNET uses the DISC command to close the connection.

The KERNET command is implemented on each computer using the control language of that computer's standard command interpreter. If the KERNET command is used without the correct arguments, it prompts the user to enter appropriate values.

### 6.4.3. Precautions When Logging Out

When following the procedures outlined above, especial care must be taken when logging out from the remote computer. Most of the computers on EENET have been configured to transmit a *control-T* character when a user logs out. This has the effect of closing the CASE network connection between the user's terminal and the computer, and is usually desirable.

When using Kermit, however, the user's terminal is only indirectly connected to the remote computer. A *control-T* character sent by the remote computer would break the link between the two computers (which is desirable). It would also be relayed, by the local computer's Kermit program (which would be in transparent or terminal emulator mode), to the link between the local computer and the user's terminal. This would have the effect of breaking the link between the user's terminal and the local computer, which is undesirable.

To avoid this problem, it is necessary either to prevent the remote computer from sending a *control-T* character, or to ensure that the local Kermit program is no longer in transparent mode when the remote computer does send a *control-T* character. An alternative form of the command used to logout from the remote computer can be provided, which delays for a few seconds before logging out. This gives the user time to switch the local Kermit program out of transparent mode before the undesired *control-T* character arrives.

### 6.4.4. **Automated File Transfer Procedures**

A simple automated file transfer procedure between the HP-A900 and VAX-750 computers has been implemented by A. M. Mvinjelwa, while an undergraduate student in the Department of Electronic Engineering [Mvin87]. This file transfer procedure uses the Connect program together with standard command files such as the TOCASE command file to establish a connection between the two computers, and specially written command files to allow the control program (running on the HP-A900) to establish an interactive session on the VAX-750, to run the Kermit program on the VAX-750 and to logout from the VAX-750. The control program performs the actual file transfers by invoking the Kermit program on the HP-A900 in such a way that it runs non-interactively.

No other automated file transfer procedures have been implemented.

## Section 7. Conclusions

The CASE DCX-850 terminal switching exchange network in use at the University of Natal, known as NUNET or EENET, links many computers, terminals and other devices. The work described in this thesis has increased the usefulness of the network in several ways which are summarised below. Some problems with the current system, as well as suggestions for improvements, are also described below.

### 7.1. Help Facility

A simple but adequate CASE Help facility has been developed, and can be accessed by all network users. This help facility uses a program running on one of the Department of Electronic Engineering's computers (the HP-A900), and two data files for help text and items of news. A terminal user can access the help facility simply by entering the name **HELP** when the network invites him to select a destination address.

The CASE help program has proved entirely satisfactory. The demand placed on the host computer (the HP-A900) is very small, even when the help facility is being used. There does not appear to be a need for a more elaborate help facility than the one presently in use.

### 7.2. Programs to Create Network Connections

The Connect program implemented on three of the Department's minicomputers has proved adequate to its task. Command files written for the program have been successfully used to create and close connections through the CASE network and to perform remote log-ins (through the CASE network) from one computer to another.

The necessity for input type-ahead buffering and XON/XOFF flow control on the host computer can prove troublesome, but the three computers on which the Connect program has been implemented manage to overcome the problems.

The Connect program cannot send a BREAK signal, nor can it control the RS-232-C control lines. The current configuration of the CASE network makes these abilities unnecessary, but they may become necessary in the future.

There is much scope for improvement in the format of the command files used by the program. A more flexible form of flow control, in addition to the GOTO commands currently used, would be especially useful.

The device locking method may be unreliable under abnormal conditions, and in any case provides no protection against non-cooperating programs. An improved method, using operating system services for file locking, device locking, and possibly for semaphore handling, would be desirable. The differences between operating systems makes this problem difficult to solve in a portable manner.

A less flexible program, named the CASE program, has been implemented on the Gerber IDS-80 computer. This program is able to create and close connections through the CASE network but does not support the flexible command files used by the Connect program.

Failures to make the connection through the network do occasionally occur. During the development of the software, failures were sometimes due to software problems, but these have now been corrected. All recent failures have been due to factors outside the control of this software. After the problem is corrected, the software is able to recover because of its retry mechanism.

The most common reasons for failure are that the specified network destination is either unavailable or nonexistent, or the communications line on the local computer is configured incorrectly, or the local communications device is locked. The most common causes of a device lock remaining after it should have been released are that a user aborted the Connect program without allowing it to clean up, or a user aborted some other program (such as a print spooler) or command file (such as KERNET or CPLOT) after it had made a connection but before it closed the connection.

## 7.3. Access to Peripherals Through the Network

Software has been written to allow some of the computers in the Department of Electronic Engineering to access peripheral devices, such as printers and plotters, that are connected to the CASE network. User-transparent methods of accessing printers connected to the network have been provided on three of the Department's computers: the Gerber IDS-80, the HP-9000 and the HP-A900. A transparent method of accessing plotters connected to the network has been provided on the Gerber IDS-80. Simple commands, available on four of the Department's computers, allow connection through the network to peripheral devices that are not supported in a user-transparent fashion.

Personal computers (PCs) connected to the network can connect to peripherals through the network by using standard communications programs, or terminal emulators.

A customised print symbiont program for the VAX-750 could be written, to provide transparent access to printers connected through the CASE network. The effort required for this does not appear to be warranted.

Data loss is possible if a connection between a computer and a peripheral device is closed while some data is still in the network's buffers and has not yet reached the device. This problem is inherent in the CASE network itself, and cannot be properly solved with existing hardware. A long delay (60 to 90 seconds) is used just before disconnecting, to guard against this problem. The delay allows data buffered by the network a chance to reach the destination device before the connection is closed. The delay is annoying, especially when one print job is closely followed by another, and in any case cannot guarantee that there will be no data loss.

## 7.4. File Transfer Through the Network

File transfers between computers connected to the CASE network may be performed using the widely available Kermit file transfer protocol and various commands that facilitate the creation of connections through the network.

PCs connected to the network can use standard implementations of the Kermit protocol to connect to other computers through the network and to transfer files.

Automated file transfers have been provided between two of the Department's computers. This work was performed by A. M. Mvinjelwa [Mvin87].

## 7.5. Data Loss When Disconnecting

When a computer sends a large quantity of data through the CASE network to a peripheral, there is no way for the computer to be sure that all the data has reached its destination, and is not still being buffered by the network, pending transmission to the destination device. If the computer closes the connection while some of the data is still in transit through the network (which possesses fairly large buffers), then that data will be lost.

Currently, in order to reduce the probability of data loss from this cause, the computer delays a long time (60 to 90 seconds) after sending all its data, before closing the connection. This delay allows the contents of the network buffers to be transmitted to the peripheral device. This procedure has two disadvantages: Firstly, no matter how long the delay, there is no guarantee that data will not be lost. Secondly, the delay that is actually used is frequently much longer than was actually required, which causes unnecessary idle periods before the device can be used again.

A possible way in which this problem can be solved is with some form of handshake between the computer and the peripheral device. The computer could send a special character sequence to the device, and wait for a response. When the computer receives a response, it can be sure that the special character sequence, and all the data that preceded it, has been received by the device.

Because a handshake facility of this nature is not included in all peripheral devices, specially designed adaptor devices could be connected between the peripheral device and the network. The adaptor hardware would have to implement the handshake protocol. It might be useful to place similar special adaptor hardware between the computer and its network connection. If this were done, the handshake protocol could be made almost entirely transparent to both the computer and the peripheral device. (The computer would need some way of informing the special hardware that it wishes to close the connection.)

## 7.6. Inability to Transmit Certain Characters as Data

Most of the ports on NUNET use XON/XOFF flow control. The ASCII *DC1* and *DC3* characters (also known as *XON* and *XOFF*, or *control-Q* and *control-S*) cannot be transmitted as data between two ports, if either of these ports uses XON/XOFF flow control. This restriction applies to DC1 and DC3 characters whose parity bit is either zero or one, so if an 8-bit character set is used, four characters become unavailable.

Many of the ports on EENET use the ASCII *DC4* character (also known as *control-T*) to signal either a *connect event*, a *disconnect event*, or both connect and disconnect events. The DC4 character (with either zero or one in its parity bit) cannot be transmitted as data between two ports, if either port uses it to signal a disconnect event.

Either of the solutions proposed below would enable any character to be sent as data through the network.

The first possible solution would be to use out-of-band signaling, also known as hardware handshake, for both flow control and the generation of connect and disconnect events. Viewed at the network end of the link between the network and a DTE, typical ports would require the following four handshake lines in addition to the two data lines and a ground line currently used: DTR (for the computer or peripheral to signal whether it is ready to receive data), CTS (for the network to signal whether the computer or device is allowed to send data), RTS (for the computer or device to signal its desire to initiate or close a connection) and DCD (for the network to signal the computer or device when a connection is made or broken). A problem with this method is that some computers cannot control or use the RS-232-C handshake lines.

The second possible solution would be to use special hardware devices between each computer or peripheral device and its network connection. These hardware devices could implement a protocol that allows the DC1, DC3 and DC4 characters to be sent as data if required, without needing the extra handshake lines required by the method described above. Protection against data loss could be built into the same hardware devices. Error detection or correction protocols with automatic retries could also be implemented. All the above functions could be made almost entirely transparent to the computer or peripheral device, but there would have to be some way of informing the special hardware when a connection is to be opened or closed.

## Appendix A1. The ASCII Character Set

ASCII is a seven-bit character set. Computers frequently store or transmit characters using eight bits, in which case the eighth bit is often either set to zero or used as a parity check. The following table shows how characters in the ASCII character set are coded. Columns **D** and **H** are the decimal and hexadecimal codes for the characters, respectively. Control characters are listed with their standard abbreviations.

| D | H | | D | H | | D | H | | D | H | | D | H | | D | H | | D | H | | D | H | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 00 | NUL | 16 | 10 | DLE | 32 | 20 | SP | 48 | 30 | 0 | 64 | 40 | @ | 80 | 50 | P | 96 | 60 | ` | 112 | 70 | p |
| 1 | 01 | SOH | 17 | 11 | DC1 | 33 | 21 | ! | 49 | 31 | 1 | 65 | 41 | A | 81 | 51 | Q | 97 | 61 | a | 113 | 71 | q |
| 2 | 02 | STX | 18 | 12 | DC2 | 34 | 22 | " | 50 | 32 | 2 | 66 | 42 | B | 82 | 52 | R | 98 | 62 | b | 114 | 72 | r |
| 3 | 03 | ETX | 19 | 13 | DC3 | 35 | 23 | # | 51 | 33 | 3 | 67 | 43 | C | 83 | 53 | S | 99 | 63 | c | 115 | 73 | s |
| 4 | 04 | EOT | 20 | 14 | DC4 | 36 | 24 | $ | 52 | 34 | 4 | 68 | 44 | D | 84 | 54 | T | 100 | 64 | d | 116 | 74 | t |
| 5 | 05 | ENQ | 21 | 15 | NAK | 37 | 25 | % | 53 | 35 | 5 | 69 | 45 | E | 85 | 55 | U | 101 | 65 | e | 117 | 75 | u |
| 6 | 06 | ACK | 22 | 16 | SYN | 38 | 26 | & | 54 | 36 | 6 | 70 | 46 | F | 86 | 56 | V | 102 | 66 | f | 118 | 76 | v |
| 7 | 07 | BEL | 23 | 17 | ETB | 39 | 27 | ' | 55 | 37 | 7 | 71 | 47 | G | 87 | 57 | W | 103 | 67 | g | 119 | 77 | w |
| 8 | 08 | BS | 24 | 18 | CAN | 40 | 28 | ( | 56 | 38 | 8 | 72 | 48 | H | 88 | 58 | X | 104 | 68 | h | 120 | 78 | x |
| 9 | 09 | HT | 25 | 19 | EM | 41 | 29 | ) | 57 | 39 | 9 | 73 | 49 | I | 89 | 59 | Y | 105 | 69 | i | 121 | 79 | y |
| 10 | 0A | LF | 26 | 1A | SUB | 42 | 2A | * | 58 | 3A | : | 74 | 4A | J | 90 | 5A | Z | 106 | 6A | j | 122 | 7A | z |
| 11 | 0B | VT | 27 | 1B | ESC | 43 | 2B | + | 59 | 3B | ; | 75 | 4B | K | 91 | 5B | [ | 107 | 6B | k | 123 | 7B | { |
| 12 | 0C | FF | 28 | 1C | FS | 44 | 2C | , | 60 | 3C | < | 76 | 4C | L | 92 | 5C | \ | 108 | 6C | l | 124 | 7C | \| |
| 13 | 0D | CR | 29 | 1D | GS | 45 | 2D | - | 61 | 3D | = | 77 | 4D | M | 93 | 5D | ] | 109 | 6D | m | 125 | 7D | } |
| 14 | 0E | SO | 30 | 1E | RS | 46 | 2E | . | 62 | 3E | > | 78 | 4E | N | 94 | 5E | ^ | 110 | 6E | n | 126 | 7E | ~ |
| 15 | 0F | SI | 31 | 1F | US | 47 | 2F | / | 63 | 3F | ? | 79 | 4F | O | 95 | 5F | _ | 111 | 6F | o | 127 | 7F | DEL |

Characters whose hexadecimal codes are 00 to 1F, as well as character 7F, are control characters. Character 20 is a blank space. Characters 30 to 39 are the decimal digits. Characters 41 to 5A are upper-case alphabetic characters. Characters 61 to 7A are lower-case alphabetic characters. All other characters are punctuation marks (codes 21 to 2F, 3A to 40, 5B to 60 and 7B to 7E).

The control characters with hexadecimal codes 00 to 2F are often referred to as **control-@, control-A, control-B** etc., up to **control-_** (control-underscore). This is because these characters can be generated on many keyboards by pressing a special control key while typing the characters @, **A**, **B**, etc. These characters are often depicted as ^@, ^A, etc. The DEL character (hexadecimal code 7F) is sometimes depicted as ^?.

The control characters have the following names:

| | |
|---|---|
| NUL | Null. |
| SOH | Start of heading. |
| STX | Start of text. |
| ETX | End of text. |
| EOT | End of transmission. |
| ENQ | Enquiry. |
| ACK | Acknowledge. |
| BEL | Bell. |
| BS | Backspace. |
| HT | Horizontal tab. |
| LF | Line feed. |
| VT | Vertical tab. |
| FF | Form feed. |
| CR | Carriage return. |
| SO | Shift out. |
| SI | Shift in. |
| DLE | Data link escape. |
| DC1 | Device control 1. Often called XON. |
| DC2 | Device control 2. |
| DC3 | Device control 3. Often called XOFF. |
| DC4 | Device control 4. |
| NAK | Negative acknowledge. |
| SYN | Synchronous idle. |
| ETB | End of transmission block. |
| CAN | Cancel. |
| EM | End of medium. |
| SUB | Substitute. |
| ESC | Escape. |
| FS | File separator. |
| GS | Group separator. |
| RS | Record separator. |
| US | Unit separator. |
| SP | Space. |
| DEL | Delete. |

## Appendix A2. EIA RS-232-C and CCITT V.24 Control Signals

The table below lists the names of the signals used in the RS-232-C interface [EIA69], together with their V.24 counterparts and the pin numbers used on the connector.

The column marked **From** indicates whether the DCE or the DTE is the source of the signal. The column marked **EIA** contains the signal's standard abbreviation and the column marked **V.24** indicates the corresponding signal in the CCITT V.24 standard. The column marked **Abbr** contains the abbreviation commonly used to refer to the signal, although this abbreviation does not form part of the RS-232-C standard.

| Pin | From | V.24 | EIA | Abbr | Name |
|---|---|---|---|---|---|
| 1 | | 101 | AA | | Protective ground. |
| 2 | DTE | 103 | BA | TXD | Transmitted data. |
| 3 | DCE | 104 | BB | RXD | Received data. |
| 4 | DTE | 105 | CA | RTS | Request to send. |
| 5 | DCE | 106 | CB | CTS | Clear to send. |
| 6 | DCE | 107 | CC | DSR | Data set ready. |
| 7 | | 102 | AB | | Signal ground. |
| 8 | DCE | 109 | CF | DCD | Received line signal detector. (Also known as Data Carrier Detect.) |
| 9 | | | | | Reserved for Data Set testing. |
| 10 | | | | | Reserved for Data Set testing. |
| 11 | | | | | Unassigned. |
| 12 | DCE | 122 | SCF | SDCD | Secondary received line signal detector. (Also known as Secondary Data Carrier Detect.) |
| 13 | DCE | 121 | SCB | SCTS | Secondary clear to send. |
| 14 | DTE | 118 | SBA | STXD | Secondary transmitted data. |
| 15 | DCE | 114 | DB | | Transmission signal element timing. |
| 16 | DCE | 119 | SBB | SRXD | Secondary received data. |
| 17 | DCE | 115 | DD | | Receiver signal element timing. |
| 18 | | | | | Unassigned. |
| 19 | DTE | 120 | SCA | SRTS | Secondary request to send. |
| 20 | DTE | 108.2 | CD | DTR | Data terminal ready. |
| 21 | DCE | 110 | CG | | Signal quality detector. |
| 22 | DCE | 125 | CE | RI | Ring indicator. |
| 23 | DTE | 111 | CH | DRS | Data signal rate selector. Source is |
| | DCE | 112 | CI | DRS | either DTE or DCE. |
| 24 | DTE | 113 | DA | | Transmit signal element timing. |
| 25 | | | | | Unassigned. |

## Appendix A3. The CASE Network

This appendix describes the features of the CASE network in detail. Most of the information here applies to any network of CASE DCX-850 nodes, but some information is specific to NUNET or EENET.

### A3.1. Basic Features

The CASE network allows many serial (RS-232-C) lines to be interconnected in a way defined either statically, by the network supervisor, or dynamically, based on requests from the devices connected to the network.

The network consists of several *nodes*, each of which contains a CASE *DCX-850* controller [CASE-850]. Each DCX-850 node has the features of a terminal switching exchange, and the nodes are interconnected in such a way that the entire network appears to the user to be a large terminal switching exchange. The DCX-850 is based on the DCX-840 [CASE-840], which does not allow port users to request connections to other ports.

Each node has several low speed RS-232-C ports, connected to computers, terminals or other devices. Nodes are interconnected by high speed composite links with an *Automatic Repeat Request* (*ARQ*) protocol, based on the *High-level Data Link Control* (*HDLC*) procedure of CCITT recommendation X.25. Low speed ports are controlled by *LSC* (*Low Speed Channel*) cards connected to the DCX-850. Low speed ports may also be connected to terminal concentrators whose composite links are connected directly to the the DCX-850.

### A3.2. Types of Ports and Connections Between Them

Each port on the network belongs to one of three types: *Internally Mapped Port* (*IMP*), *User Mapped Port* (*UMP*) or *Automatically Mapped Port* (*AMP*). (Almost all the ports on NUNET are UMPs.)

An IMP port is connected permanently to another IMP. This connection is set up by the network supervisor. If the two ports are on different nodes of the network, the route used by the connection may automatically be changed, to accommodate link failures. (When a connection between non-IMP ports is broken due to a link failure, it is not re-routed in this way.)

An UMP port can be connected to any other UMP. When a connect event occurs on an UMP that is not connected to any destination, the network outputs a message inviting the user to select a destination. A destination address is then entered by the user, and the network attempts to connect the port to the destination. Success or failure is indicated by a message sent by the network to the port. If the connection is successful, the message COM will be given. Other possible messages are listed in section A3.9. The destination requested by the user must be another UMP (or a group address or alphanumeric name that maps to an UMP). Once a connection has been made, a disconnect event originating from either port will break the connection.

An AMP port is not permanently connected to any destination, but is automatically connected to a predetermined destination when a connect event occurs. The destination used for this automatic connection is not supplied by the port user, but is a group address specified the network supervisor. When a connect event occurs, the network automatically attempts to connect the AMP to a port within the relevant group. As is the case for UMP connections, the network outputs a message indicating success or failure of the connection attempt.

## A3.3. Port Addresses, Group Addresses and Alphanumeric Names

Each port on the network is identified by a *port address*, in the form *NN*.*PPP*, where *NN* is the node number and *PPP* is the logical port number on that node. This is the lowest level of addressing that can be used when a connection between two ports is made.

The supervisor at each node may define several *group addresses*, also known as *short form addresses*. Each group address translates to a list of one or more port addresses. When a group address is given as the destination for a connection request, it is specified in the form :*GG* or **GG*, where *GG* is the group number. The network tries to connect to each of the ports in the group, until either a connection is made successfully or the list is exhausted. This allows any suitable port to be selected automatically by the network when a group address is specified as a connection destination. The order in which the ports are specified in the group definition determines the order in which they will be checked.

When the group definition indicates that a port on a remote node may be used, the local node requests the remote node to make the connection. In this case, the remote node ignores the port number that was specified in the local node's group definition, and tries only the ports that are named in its own definition for the relevant group. This allows group address definitions on a particular node to be modified without the change having to be reflected on all the nodes in the network.

For example, if nodes 1 and 2 have the following definitions for group number 5:

| | |
|---|---|
| Node 1: | 01.002-005 02.008 |
| Node 2: | 02.024 01.001 02.032 03.001 |

and if a user on node 1 requests to connect to group address 5, then the following ports will be tried, in this order:

01.002, 01.003, 01.004, 01.005, 02.024, 02.032.

Note that port 02.008 was not tried, although the definition on node 1 suggested that it would be. Also, no ports on node 3 were tried, although they would be tried if a user on node 2 requested to connect to group address 5.

The relationship between group addresses and port addresses is a many-to-many mapping. Each group address may translate to a list of several port addresses, and each port may be included in several groups.

In addition to group addresses, each node may define several *alphanumeric names*, each of which corresponds to a single group address. An alphanumeric name is simply a series of up to eight digits and upper-case letters. When a user gives an alphanumeric name as the destination for a connect request, the operation is exactly the same as if the corresponding group address had been given.

### A3.4. Connections and Routing Between Nodes

A high speed composite link connection between two nodes is called a *link*. Each link requires its own ARQ card on the DCX-850. A number of logical channels are carried on the same link. There must be a sufficient number of these channels to allow for the maximum number of inter-node connections that may be in use at any time.

One or more links between the same pair of nodes constitutes a *pipe*. The traffic on a pipe is shared among the links that make up the pipe. When a connection is attempted using a particular pipe, the link most capable of handling the increased load is chosen, but if that link fails, other links in the same pipe will not be tried.

Each node has a *route* table that defines how other nodes may be accessed. For each remote node, the route table contains a list of pipes that may be used for a connection to that node. The traffic load is not shared between pipes. A connection between one node and another uses the first operational pipe listed under the destination node's entry in the source node's route table.

Each node must have a route to all other nodes in the network. If a destination node is not connected to the source node by a direct link, but must be accessed via intermediate nodes, then the route to the destination node would specify a pipe that is connected to the first intermediate node. Subsequent hops in the routing of a connection to the destination node would be determined by the route tables in the intermediate nodes.

### A3.5. Low Speed Channel Flow Control

The CASE network supports two varieties of flow control [CASE-LSC] [CASE-815]. The first, *terminal flow control* allows a device connected to a port on the network to signal whether or not it is ready to receive data. The second variety, *buffer overflow protection*, allows the network to signal whether or not it is ready to receive data.

These two varieties of flow control can be configured independently. Each may be disabled, set to use in-band signaling (XON/XOFF), or set to use out-of-band signaling (using the *DTR* control line for terminal flow control, and the *CTS* control line for buffer overflow protection). Usually, a particular port will have both varieties of flow control set in the same way, but this is not essential. All four ports on a particular LSC card plugged into a DCX-850, or all eight ports on a DCX-815 terminal concentrator, share a single flow control configuration.

Data transmission between one port and another is affected by the terminal flow control configuration at the destination port, and by the buffer overflow protection setting at the source port. If, for example, the source port uses out-of-band buffer overflow protection and the destination uses in-band terminal flow control, then either an XOFF character received from the destination port or an impending network buffer overflow will result in the CTS signal to the source port being turned off.

In-band flow control has the advantage that only a three-wire connection is required (for ground, transmitted data and received data). It has the disadvantage that the XON and XOFF characters cannot be sent as data, because they are intercepted by the network and treated in a special way.

Out-of-band flow control has the advantage that any data characters may be transmitted. It has the disadvantage that more wires are required for the connection between a device and the network controller. Two extra wires would be required for the DTR signal from the device and the CTS signal from the network. If out-of-band signaling is used for flow control, it is likely that out of band signaling will also be used to signal connect and disconnect events. This would require at least one additional wire, for the device to signal the network of its intention to connect or disconnect. Another wire for the network to inform the device of the connection status would often be desirable. A connection using out-of-band signaling would therefore usually use at least seven wires (four handshake lines, two data lines and a ground).

Most of the ports on EENET have both the terminal flow control and buffer overflow protection features configured for in-band (XON-XOFF) flow control.

## A3.6. **Connect and Disconnect Events**

An AMP or UMP port must generate a *connect event* before the network will connect it to a destination. In the case of an AMP, the network attempts to connect the port as soon as the connect event occurs. In the case of an UMP, a connect event causes the network to print a prompt inviting the port user to select a destination address before the connection can be attempted. The user may respond to this message by typing the desired destination address and a carriage-return, or may request that a previous unsuccessful connection attempt be queued for automatic retry by typing Q and a carriage-return. The network will respond with one of the messages described in section A3.9.

Once a connection has been established, a disconnect event generated by either port will cause the connection to be closed.

The network supervisor may specify the nature of the connect and disconnect events for each port. For UMPs, a connect event requires that the condition mentioned below be followed within about 10 seconds by a carriage-return character. For AMPs, the condition mentioned below immediately causes an attempt to connect to the required destination. A disconnect event causes an immediate disconnection.

The handshake lines referred to in the descriptions below depend on the configuration of the LSC card, which may be either DTE (Data Terminal Equipment) or DCE (Data Communications Equipment). A DCE version of the LSC card would generally be used if the devices connected to it were terminals or computers. A DTE version would be used if devices such as modems were connected to it. The two configurations are identical internally; the only differences are in the labels associated with various indicators on the front panel, and in a cable between the card itself and the external plug seen by the device.

For connect events, the following options are available. The numbers correspond to the codes that would be used by the network supervisor to select these options.

0) The signal on channel pin number 20 becoming true. On a DCE version LSC card, this is the Data Terminal Ready (DTR) signal from external pin number 20.

1) The signal on channel pin number 4 becoming true. On a DTE version LSC card, this is the Data Carrier Detect (DCD) signal from external pin number 8. On a DCE version LSC card, this is the Request To Send (RTS) signal from external pin number 4.

2) A *break* signal on the received data line generates a connect event.

3) The port cannot generate a connect event. It may however be used as the destination for a connection initiated by another port.

4) This option is designed for use with dial-in modems. A connect event is generated if the signal on channel pin number 25 becomes true, followed within 10 seconds by the signal on channel pin number 4 becoming true. On a DTE version LSC card, this is the Ring Indicator (RI) signal from external pin number 22, followed by the Data Carrier Detect (DCD) signal from external pin number 8.

5) An ASCII DC4 character (also known as a *control-T* character) received by the port will generate a connect event.

A disconnect event results in the current connection being closed and all the port's handshake signals being made false, for a period of at least one second. Any data that has been received from one port but is still in the network's buffers and has not been sent to the other port, is discarded when a disconnect event occurs. The following disconnect events are available. The numbers used here correspond to those that would be used by the network supervisor when the port is configured.

0) The signal on channel pin number 20 becoming false. On a DCE version LSC card, this is the Data Terminal Ready (DTR) signal from external pin number 20.

1) The signal on channel pin number 4 becoming false. On a DTE version LSC card, this is the Data Carrier Detect (DCD) signal from external pin number 8.

2) A *break* signal on the received data line generates a disconnect event.

3) The port cannot generate a disconnect event. A connection to this port must be terminated by the other port, or by the network supervisor.

4) This is a special *dual level disconnect*, designed for use with dial-in modems. A BREAK signal disconnects the current connection and immediately results in an invitation to select a new destination, without the handshake signals being dropped. If the signal on channel pin number 4 becomes false, an ordinary disconnect event occurs. This includes dropping the handshake signals in the usual way. On a DCE version LSC card, channel pin number 4 is the Data Carrier Detect (DCD) signal from external pin number 8.

5) An ASCII DC4 (control-T) character received by the port will generate a disconnect event.

On EENET (node 8 of NUNET), various types of ports are usually configured for the following connect and disconnect events. There are some exceptions to this scheme.

Terminals and PCs: BREAK connect event (option number 2 in the list above). DC4 disconnect (option number 5).

Minicomputers: DC4 connect event (option 5) or no connect event (option 3). DC4 disconnect event (number 5).

Dial-in modems: RI and DCD connect event (option 4). DCD disconnect event (option 1). (The dual-level disconnect feature, option 4, is not used.)

Other devices: No connect event (option 3). No disconnect event (option 3).

## A3.7. Data Parity

Although some other options may be selected, the network usually works with either seven data bits plus a parity bit, or with eight data bits. When data is transmitted from one port to another, all eight bits are transmitted unmodified. This allows the devices at either end of the connection to decide for themselves which parity convention to adopt. The network cannot translate one form of parity checking to another.

The configuration of each port includes the type of parity to be used for system messages to the port. Four options are available: Odd parity, even parity, mark parity (high bit always 1) and space parity (high bit always zero). This parity setting affects error messages and other messages from the network. It also affects XON and XOFF characters if the port is configured for in-band buffer overflow protection. It does not affect data sent from the device at the other end of a connection.

During the connection dialogue between an AMP and the network, the parity of transmissions from the device is ignored. Transmissions to the device use the parity specified in the port's configuration.

## A3.8. Transmission Baud Rate

The baud rate of each port may be configured individually. When two ports with different baud rates are connected, the network automatically converts between the two rates. Each port may also have the speed matching feature disabled, in which case it cannot be connected to another port with a differing baud rate.

Some low speed ports support Automatic Baud rate Recognition (ABR). These ports can be configured to recognise the baud rate used when a connect event occurs.

## A3.9. System Messages

When an UMP generates a connection event, the network responds with a message that usually consists of three lines. The first and last lines contain messages defined by the network administrator, and the second line contains the port address of the port involved. The first line is intended for a message identifying the network, and the third line is intended for an invitation to select a destination. Either or both of these message lines may be undefined, reducing the entire message to one or two lines, instead of three. This message cannot be disabled.

When a connection from an AMP or UMP is attempted. the network responds with a short message indicating success or failure. The following messages are possible here, and cannot be disabled [CASE-850]:

| | |
|---|---|
| COM | Connection successful. If this message occurs after a "Q" request, a BEL character will be output before and after the message. |

| | |
|---|---|
| **MOM** | Connection might be successful or unsuccessful, wait for another message. |
| **ERR** | This is usually due to the presence of illegal characters (such as lower-case letters) in the destination name. It can also occur if the originating port has been closed by the network administrator. |
| **INV** | Invalid request. This could be because the destination port is set to an incompatible speed or is not an UMP. It also occurs when an invalid queue request is given. |
| **NP** | No such port. |
| **OCC** | Destination port is busy, or all suitable ports in destination group are busy. |
| **DER** | Destination is closed or out of order, or a connection to a remote node cannot be made. |
| **NA** | Access not allowed, due to restrictions imposed by the network administrator. |
| **NC** | No free channels on link between nodes, or composite link serving either port is overloaded. |
| **QUEUED** | The connection attempt has been queued for automatic retry. Wait for another message, which will be COM, NP, NA, INV or DER. |

When a connection is broken, the message DISC is sent to both ports, preceded and followed by a BEL character. The DISC message may be disabled.

Other messages [CASE-850] [CASE-815], some of which may be disabled, include:

| | |
|---|---|
| **... OVFL** | The USO working memory is almost fully committed. This problem should automatically clear after a short time. |
| **CNX FAILURE PLEASE REQ RECONNECTION** | An inter-node link failure caused the connection to be broken. |
| **DATA LOST** | Data has been lost. |
| **LINK DOWN** | The link between a terminal concentrator and the DCX-850 has been broken. |

**LINK UP** The link between a terminal concentrator and the DCX-850 has been restored.

## A3.10. Queueing an Unsuccessful Connection Attempt

If a connect attempt fails with an OCC or NC error, the network may be requested to queue the request, if this facility has not been disabled by he network supervisor. On an UMP port, this is done by entering a Q character, followed by a carriage-return, in response to the next network prompt. On an AMP port, queueing is automatic.

If the request is successfully queued, the message **QUEUED** will be output. When the connection is eventually made, a **COM** message, preceded and followed by a *BEL* character, will be output. A disconnect event generated by the device connected to the port will cause a queue request to be cancelled.

## A3.11. Access Control

Each UMP port may be allowed access to only certain destination addresses. One of the following eight access levels may be selected by the network supervisor, for each UMP port.

0) No restrictions. The port may connect to any other UMP.

1 to 6) A user at this port may not specify a port address as a destination. A group address or an alphanumeric name must be used. The second last digit of the group address specified (or the group address corresponding to the alphanumeric name specified) must be either 0 or the same as the access level.

7) The port may connect to any group address or alphanumeric name. It may not connect to a port address.

A consequence of this arrangement is that group addresses 0 to 9, 100 to 109 and 200 to 209 may be accessed by any UMP. Group addresses 70 to 99 and 170 to 199 may be accessed only by UMPs with access level code 0 or 7. Other group addresses may be accessed by UMPs whose access level is 0, or 7, or the appropriate value in the range 1 to 6. Ports that do not belong to any group can be accessed only by UMPs with access level 0.

Most of the user terminals on EENET are set to access level 7. This means that they can connect to any group address or alphanumeric name, but cannot connect to port addresses. A few terminals, and all the computers that are able to initiate connections, are set to access level 0. This allows them to connect to any port or group address.

## Appendix A4. Kermit

Kermit is a file transfer protocol developed at the University of Columbia [Kerm84] [Kerm85]. It was originally intended for file transfers between microcomputers (PCs) and minicomputers or mainframes, but implementations of the Kermit protocol are now available for more than 100 systems, either from the University of Columbia or from various user groups and networks.

### A4.1. How Kermit Works

Kermit is designed for the transfer of sequential data files over serial links. The data is transmitted in the form of packets. Each packet consists of a header that identifies the type, length and serial number of the packet, and ends with a checksum or CRC code (for error detection). Up to 91 bytes of data are included in the packet. The basic format of a packet is shown in figure A4.1.

The first character of the packet, called the MARK, is usually an ASCII SOH character (character number 1, or control-A). The particular character that is used may be changed, provided both sides agree. The MARK character is the only non-printable character that must be transmittable between the two computers. All other data is encoded in a printable way.

The LEN character gives the length of the packet. This may vary from "#", meaning 3, to "~", meaning 94. The number represented by the character is 32 less than the numeric value of the character's ASCII code. The length specified here excludes the MARK and the LEN bytes, but includes the SEQ and TYPE bytes, the data, and the CHECK.

The SEQ byte represents a sequence number, between 0 and 94, encoded as a printable character from a blank space (meaning 0) to "~" (meaning 94). The sequence number is incremented for each packet, and wraps around when necessary.

The TYPE byte represents the type of packet. The packet types include Send-Init (for initialisation), File header, Data, Acknowledgement (ACK), Negative acknowledgement (NAK), End of file, and End of group. There are also other packet types.

A packet's data field is optional. If it exists, the data is encoded in a printable way. This encoding method uses a special prefix character (usually "#") to indicate control characters. Another prefix character (usually "&") may be used to indicate a byte whose eighth bit is set to 1, but some Kermit implementations do not allow this. A third special prefix character (usually "~") is often supported for encoding groups of identical characters, to shorten transmission time.

The CHECK is a one- or two-byte checksum, or a three-byte CRC code. The communicating systems have to agree on the type of checking used.

The transmitting Kermit program requires an acknowledgement for every packet it sends, and will re-send the packet if it is not acknowledged within a suitable period (provided the computer system allows timeouts of this nature).

| MARK | LEN | SEQ | TYPE | Data | CHECK |
|---|---|---|---|---|---|
| 1 byte | 1 byte | 1 byte | 1 byte | Variable length | 1, 2 or 3 bytes |

**Figure A4.1** Structure of a Kermit packet.

## A4.2. **The Basics of Using Kermit**

Using the Kermit program is most easily described for the case of a file transfer between a PC and a larger minicomputer or mainframe. The PC needs a serial communications port which can be connected to the host computer. The host computer will then treat the PC as an ordinary terminal, which can be used for an interactive session.

Once the Kermit program is running on the PC, the user will have to use his single keyboard and screen to communicate with two computers -- the PC and the remote host. The Kermit program usually provides two modes, to allow the screen and keyboard to be switched between communicating with the PC and communicating with the remote host computer.

At first, commands entered on the keyboard affect the local PC. A special command, usually called CONNECT, switches to a transparent mode. In this mode, characters typed on the PC keyboard are relayed to the remote computer (which treats them as if they had been typed on an ordinary terminal), and information sent from the remote computer (which would have appeared on the screen of an ordinary terminal) appears on the PC screen.

Some method of terminating this transparent mode is required. The Kermit program on the PC typically provides a special character, which is known as the *ESCAPE* character but is unrelated to the ASCII ESC character, for this purpose. The ESCAPE character is usually *control-]* (*control-right-square-bracket*). When the user types the ESCAPE character, the Kermit program waits for another character to be typed. Depending on the second character typed, one of several things could happen. Usually, if the second character is also the

ESCAPE character, then a single ESCAPE character will be sent to the remote computer. If the second character is a "C" then the transparent mode will be terminated, and the PC keyboard and screen are again used to communicate with the local PC. There are often other permissible values for the character typed just after an ESCAPE character.

Because the user is able to communicate with either the local PC or the remote computer, he can log-in to the remote computer and start running the Kermit program on that computer. Once a Kermit program is running on both computers, the user can transfer files between the computers. When the file transfers are complete, the user can terminate the Kermit program on the remote computer, and logout. He may then terminate the Kermit program on the local PC.

In the above description, the Kermit program on the PC is said to be running in *local mode*. It uses a communications line to perform file transfers and uses the PC's screen and keyboard to communicate with the user. The Kermit program on the remote computer is said to be used in *remote mode*. It performs file transfers on the same serial communications line that it uses to communicate with the user.

Provided a few requirements are met, file transfers between two minicomputers or mainframes using Kermit is similar to that between a PC and another computer. It requires one of the computers to have both a terminal line that is used for typing commands, and a serial line that can be linked to the other computer. The terminal used by the user is similar in concept to the screen and keyboard of a PC. The other serial line corresponds to the PC's serial line. The Kermit program on the computer that the user's terminal is connected to must have a transparent connection mode, and must be capable of transferring files on a communications line that is distinct from that used for communicating with the user. Many Kermit implementations on minicomputers and mainframes lack this capability, but it is becoming more common.

### A4.3. Kermit Commands

Not all Kermit implementations use the same commands, but those described below are almost universal.

**SEND** *filespec*

Send the specified file or group of files. Wild-card characters can often be used in the filespec. Options are sometimes provided to send the file under a different name.

**RECEIVE**

Receive a file or group of files from the other computer. No file name is needed because the other Kermit sends a packet containing the name. Options are sometimes provided to rename the file as it is received.

To send files from the local computer to the remote computer, the remote computer must be told to RECEIVE before the local computer is told to SEND. This is done by switching between transparent mode and command mode on the local Kermit, whenever necessary. To send a file from the remote computer to the local computer, the remote computer is told to SEND immediately before the local Kermit is switched from transparent mode to command mode. The local computer is then told to RECEIVE.

**CONNECT**

This is the command usually used to switch the local Kermit program into transparent mode. Transparent mode is usually terminated by typing *control-]* (control right-square-bracket), followed by the letter C.

**SET**

This command may be used to configure the Kermit program in various ways.

**SHOW**

Displays the value of the SET parameters.

**EXIT**

Exit from the Kermit program.

## A4.4. **Kermit Servers**

Some Kermit programs have a server mode, which may be entered when the Kermit program is in remote mode (i.e., is running on the computer other than the one to which the user's keyboard or terminal is connected). Server mode allows the remote Kermit program to respond to commands received as packets from the local Kermit program. After starting the server, the user no longer has to switch in and out of the local Kermit's transparent mode in order to give commands to the remote Kermit program.

Some special commands need to be provided by the local Kermit program if the user will communicate with a remote Kermit server. The following commands are commonly used when the remote Kermit is in server mode. Note that all these commands are given to the local Kermit program; the remote Kermit program is told what to do by packetised commands sent by the local Kermit.

SEND *filespec*

This is identical to the SEND command used without server mode. It sends files from the local computer to the remote server.

GET *filespec*

Transfer a file or group of files from the remote server to the local computer. This command, given to the local Kermit program, performs the task that would be accomplished (without a server) by giving a SEND command to the remote Kermit and a RECEIVE command to the local Kermit.

**FINISH**

Terminate the remote server. This may terminate the remote Kermit entirely, or may simply switch it from server mode back to normal mode. After giving this command, a user would usually switch the local Kermit into transparent mode, logout from the remote computer, terminate the transparent mode, and then terminate the local Kermit program.

LOGOUT

LOGOFF

Terminate the remote server, as for the FINISHED command, and also terminate the interactive session on the remote computer.

**BYE**

Terminate the remote server and the interactive session on the remote computer, as for the LOGOUT command, and also terminate the local Kermit program. Equivalent to LOGOUT followed by an EXIT command.

# Appendix A5. CASE Network Help Facility

This appendix includes detailed information about the CASE network help facility, including the format of its data files, its installation procedure, and error handling.

## A5.1. Format of Help and News Files

The Help and News files used by the Case_help program contain information that will be sent to the user when he requests Help or News. The contents of the relevant file are output line by line, until either a line beginning with a dot (.) is found, or twenty lines (nearly a full screen on many terminals) have been output since the last pause. When the program pauses, the user is prompted to type any character to continue, an **A** to abort or an **R** to restart from the HELP/NEWS/ABORT prompt. The prompt line is erased after the user has responded, so a line in the help or news file that begins with a dot will finally appear as a blank line on the display. The files must be ordinary text files, which are called *Type 3* files on the HP-A900.

## A5.2. Installation on the CASE Network

The CASE network port connected to whichever port on the HP-A900 is assigned to the Case_help program must be given the alphanumeric name **HELP** before network users can access the help facility. The port should be configured to allow the computer to generate disconnection events by transmitting a *control-T* character (ASCII *DC3*), but the computer should not be allowed to generate a connection event. The required configuration must be performed by the network supervisor.

A *group address* must be defined and mapped to the correct port number. The alphanumeric name **HELP** must be defined and mapped to the correct group number. The configuration on our network, as reported in the network's Supervisor Monitor Mode, is as follows:

```
< NA HELP
NAME     ADDRESS
HELP     99

< AD 99
99 08.007
```

```
< PO 7
P=007
STATUS    U/A-DST    CUR-SPEED
  O          N/C         DLL
DEV:CHN PAR TYP SPD SYSM ECHO CNX/DCNX/LVL/NEU/TST I/A-DST
00:071   E   U   D   I    N     3/5/0/1/1              .
```

The above information means that the alphanumeric name HELP is associated with group number 99, and that group address 99 is associated with logical port number 08.007. Port number 7 on node 8 corresponds to physical channel number 071 on device 00, and is configured as an UMP that cannot generate a connect event, but can generate a disconnect event by sending a *control-T* character.

## A5.3. Installation on the Host Computer

Whenever the HP-A900 computer is booted up, a file containing CI (Command Interpreter) commands is executed. This file is usually called **/SYSTEM/WELCOME.CMD**, or some similar name. The WELCOME file starts several programs that are needed by the operating system, and enables the terminal ports that will be used for interactive sessions. It must also start the Case_help program, with the following command:

```
RU /PROGRAMS/CASE_HELP LU helpfile newsfile
```

The *LU* number given in the command is the system logical unit number of the serial port that will be used by the help system. The *helpfile* and *newsfile* arguments are optional. If specified, they are the names of the files that contain Help and News information, respectively. If the file names are not specified as command line arguments, the files **/SYSTEM/CASE_HELP.DAT** and **/SYSTEM/CASE_NEWS.DAT** are used. The Case_help program must be compiled and the executable file placed in the **/PROGRAMS** directory for this command to work.

The Case_help program installs itself as the *primary program* [RTE-DRM] for the appropriate LU. The terminal port on the HP-A900 is configured so that any input character received while there is no pending read request cause the primary program to be scheduled. It is this feature that awakens the program when a user connects to the associated port and types any character.

Most terminal ports have a program called *PROMT* as their primary program; PROMT is (indirectly) responsible for issuing log-in prompts and system *break-mode* prompts [RTE-Ori]. The Case_help program should first be run after the terminals have been enabled. This will ensure that the terminal to be used by the Case_help program is properly configured before the program begins running. If PROMT has already been made the primary program on the port used for CASE help, the Case_help program will correct the problem, but if PROMT or some other program is made the LU's primary program after the Case_help program has started, then manual intervention is required to correct the problem.

If an LU's primary program cannot be scheduled when an unexpected input character arrives, the LU's secondary program is scheduled. If the secondary program cannot be scheduled, no further action is taken. To avoid having some other program scheduled as the LU's secondary program, the Case_help program enables a program called **NULL** as the secondary program. The Null program need not exist at all; if it does exist, it should do nothing, and could be compiled from Fortran source code containing only a *STOP* statement.

Although it should not be necessary for the NULL program to exist at all on the system, under some conditions during the development of the Case_help program it was found that unexpected input characters caused the system to generate *Illegal Interrupt* messages if the secondary program was not present. This action no longer occurs, possibly due to the installation of a new version of the operating system. If the Illegal Interrupt messages appear again in the future, the NULL program should again be used. The *Welcome* file will have to execute the command **RP /PROGRAMS/NULL** and, of course, the file **/PROGRAMS/NULL.RUN** would have to contain a suitable executable program.

## A5.4. Error Recovery and Diagnostics

When the help program is first scheduled, it checks that its first command line argument is a valid system *logical unit* number, representing the LU to be used for communication with the CASE port. The LU number must be greater than 1 and less than 256, or an error message will be printed on the terminal scheduling the program.

Once scheduled correctly the first time, the program can be re-scheduled automatically by an interrupt from the relevant LU, or manually by a user. This occurs because the program enables itself as the *primary program* for that LU [RTE-DRM], and terminates in a way that will allow it to be re-scheduled when a character is received or when the operating system RU command is used to run the program. An **EXEC (6,0,1)** system call (terminate saving resources) is used here [RTE-PRM].

If an error occurs in handling the Help or News files, or if the file type of either of these files is incorrect, then a message is printed on the system console indicating the time, file name and nature of the error. A similar message, but excluding the time and preceded by the message **<ERROR>**, is sent to the network terminal port. The help user is disconnected after this message is printed, and the program then waits for another user to connect.

Each program in an RTE-A system has a *break flag* that can be set by system commands, and can be explicitly checked and reset by the program. The break flag is often used to terminate programs that check for it. For testing purposes, the Case_help program can be made to check its break flag frequently. Some lines in the program's Fortran source code that are associated with testing, including those that check the break flag, begin with a **D** character. These lines are ignored (treated as comments) by the compiler, unless the program is compiled with a special compiler control option.

When the break flag is found to be set (provided checking is enabled), the program terminates, after printing a message on the system console, enabling the PROMT program as the LU's primary program and disabling the secondary program. If a user is connected when the break flag is seen, the message **<BREAK>** is sent to the user, who is then disconnected.

If the user takes too long to respond when the program expects input, the message **<TIMEOUT>** will be displayed and the user will be disconnected. This protects the help facility from users who leave their terminal unattended without terminating the help session properly. While a user is viewing a screenful of output, about 90 seconds is allowed before a timeout occurs.

If for any reason the primary and secondary programs associated with the HP-A900 logical unit used by the Case_help program are changed, the program will no longer be scheduled when a user connects to the relevant port and types a wake-up character. The help facility can be re-enabled by issuing the following commands:

CN *lu* 20B *helpname*

Control request number 20 (octal) sets up the primary program. *Lu* is the logical unit number used by the help system, and *helpname* is the name of the help program. The actual program name must be used here, not the name of the executable file containing the program. The program name is usually derived from the first five characters of the name of the executable file.

CN *lu* 40B NULL

Control request number 40 (octal) sets up the secondary program, which should be named NULL. This step will probably not be necessary (see section A5.3).

## A5.5. **Removing the Help Program from the Host Computer**

After it is installed, the help program will remove itself from the system if it is run by a SuperUser with the command line arguments **ST,OP**, as follows:

RU, *program* , ST, OP

Note that the name of the installed program itself must be used, not the name of the executable file containing the program. When the program is re-scheduled with its first two integer parameters set to ST and OP, it verifies that it was invoked by a SuperUser. If this is not the case, an error message is printed on the system console and on the invoking terminal. If a valid stop request is received, the program enables PROMT as the primary program on the relevant LU, disables the LU's secondary program, sends a message to the system console and terminates.

If the help program or program Null were *RP*'d as permanent programs, they will still be present (but not active) after the help program has terminated. They can be removed by the OF command if no longer required, as follows:

```
OF, program, ID
```

If the program has been compiled with checking of its break flag enabled, then another way of removing the program from the system is to set its break flag with the command

```
BR, program
```

and then to cause the program to run (if it is not doing so already), so that it will notice that its break flag is set. As soon as it sees that its break flag is set, the program will inform the current Help user, reinstate PROMT as the primary program on the LU used, disable the secondary program, write a message on the system console and terminate.

Ordinary users, as opposed to SuperUsers, can cause the Case_help program to remove itself from the system, merely by setting its break flag and either waiting for a help user to schedule it, or scheduling it themselves. Because of this, it is recommended that the program not be compiled with break flag checking enabled, except for testing purposes.

Once the program is installed, scheduling it by any means other than an interrupt from the proper LU will usually cause an error message to be printed on the scheduling terminal and the console. SuperUsers (such as the system manager) can force the program to run as though it had been scheduled by an interrupt from the appropriate LU, by giving -1 as the first command line argument passed to the program, as follows:

```
RU, program, -1
```

This feature may be used for testing.

## Appendix A6. The Condpp Conditional Preprocessor

### A6.1. Basic Features of Condpp

The Condpp program conditionally copies lines from one or more input files to an output file, under control of special preprocessor directive lines in the input files. Preprocessor symbols may be defined or undefined by directive lines or by command-line options given when the program is run. A symbol table maintained by Condpp simply lists the symbols that have been defined; unlike some other preprocessors, symbols that have been defined do not have values.

There are three classes of preprocessor directives. **$DEFINE** and **$UNDEFINE** directives define or undefine preprocessor symbols. **$IF**, **$ELSEIF**, **$ELSE** and **$ENDIF** directives determine the conditions under which lines from the input file are copied to the output file. **$COMMENT** and **$ENDCOMMENT** directives may be used to comment-out sections of an input file, preventing it from being copied to the output file. The **$C** directive is used for single-line comments, when a $COMMENT ... $ENDCOMMENT block is unnecessary.

$COMMENT ... $ENDCOMMENT blocks may be nested to any depth. $IF ... $ENDIF blocks may be nested to an installation dependent maximum depth, determined when the Condpp program itself is compiled.

The $IF and $ELSEIF directives can only test whether or not a preprocessor symbol is defined. No more advanced ability has yet been needed, but the Condpp program could be modified to handle more complicated expressions in $IF or $ELSEIF directives, if the need ever arose.

The symbol table maintained by Condpp, as has been mentioned, is simply a list of the symbols that have been defined. Due to the small size of the table, looking up a symbol is performed by a simple linear search. Upper- and lower-case characters are considered to be identical when the symbol table is searched, but symbol names are stored in the table exactly as they were specified when first defined. This is meaningful if the symbol table is printed at the end of a job.

Some preprocessor symbols are predefined by the particular version of the Condpp program. These identify the computer and operating system in use. Command-line options specified when the program is run may undefine some or all of these predefined symbols, and may define other symbols needed for processing of the input files. Command-line options also specify the names of the input and output files and control the operation of debugging modes, including printing of the symbol table at the end of the job.

### A6.2. Command Line Options

The following is a list of the command line options used by the Condpp program. Options may appear in any order. Where an option takes an argument, blank space may, but need not, separate the argument from the option letter. Each option begins with either a + or a - sign. With the exception of the **+B** and **-B** options, an option introduced by a + sign is equivalent to one introduced by a - sign. Upper- and lower-case option letters are equivalent. All arguments not introduced by + or - characters are assumed to be input files.

| | |
|---|---|
| **-O** *outfile* | Specifies the name of the output file. |
| **-V** | Enables a debugging mode, in which every line of input is copied to the output, preceded by an indication of whether that line would have been output if debug mode had not been active. (The V stands for *verbose*.) |
| **+B** | |
| **-B** | The **+B** option instructs Condpp to output a blank line for every input line that would otherwise have been excluded from the output. The **-B** option cancels the effect of a previous **+B** option. |
| **-P** | Print the symbol table at the end of the program. If this option is given without an input or output file being specified, no file processing will occur, and only the predefined symbols will be printed. |
| **-D** *symbol* | Defines the given symbol. |
| **-U** *symbol* | Undefines the given symbol. |
| **-N** | Undefines all symbols. |

The **-D**, **-U**, **-N**, **+B** and **-B** options are processed in the order in which they appear relative to the input files. This means that, if there are several input files, different options may be in effect for each one.

If no input file name is specified, input is read from the program's standard input file, which will usually be the user's interactive terminal. Similarly, if no output file name is specified, output is written to the program's standard output file. A pseudo-file name consisting just of a dash (**-**) can be used for an input or an output file name. This has the effect of making input or output use the standard I/O files. This facility is especially useful when some of the input data comes from a named file and some comes from the standard input file.

If the program is run with no arguments, a short help message is printed. A single dash (**-**) can be given as the only command line argument, if the program is required to read the standard input, write to the standard output and use only the predefined symbols.

## A6.3. **Program Operation**

The Condpp program first checks the validity of its command line options. **-O** and **-V** options take effect immediately. If a **-P** option appears, and no input or output files are specified, then no further processing is done; the table of predefined symbols is printed immediately. After checking for **-O**, **-V** and **-P** options, the program processes the command line arguments a second time, with input files and **+B**, **-B**, **-N**, **-D** and **-U** options being processed in the order they appear. If, for example, a **-N** option appears between two input file names, then the preprocessor symbol table will be cleared before the second input file is processed. After all the input files have been processed, the symbol table will be output if the **-P** option was specified. The symbol table is always output on the standard output file, even if the **-O** option is used to redirect the rest of the program's output.

As the input files are read, they are checked for preprocessor directives. Each such directive is a line whose first character is a dollar sign (**$**), but not all lines that start with a dollar sign are preprocessor directives.

The following preprocessor directives are recognised. Although the keywords are listed in upper-case below, they may appear in upper- or lower-case. Optional text may appear after the $COMMENT, $ENDCOMMENT, $ELSE and $ENDIF directives, and is ignored by the program.

| | |
|---|---|
| **$C** *text* | Single line comment. |
| **$COMMENT** | Start of a COMMENT block. Lines within a $COMMENT ... $ENDCOMMENT block are not copied to the output. These comment blocks may be nested to any depth. |
| **$ENDCOMMENT** | End of a COMMENT block. |
| **$IF** *expression* | Start of an IF block. Lines after the $IF will be copied to the output only if the expression is true. |
| **$ELSEIF** *expression* | Introduces optional ELSEIF part of an IF block. |
| **$ELSIF** *expression* | A synonym for $ELSEIF. |
| **$ELSE** | Introduces optional ELSE part of an IF block. |
| **$ENDIF** | End of an IF block. |

|  |  |
|---|---|
| **$DEFINE** ***symbol*** | Define the specified symbol. A symbol is either defined or undefined; it cannot have any other value. |
| **$UNDEFINE** ***symbol*** | Undefine a symbol. Delete the symbol's name from the symbol table. |
| **$UNDEF** ***symbol*** | A synonym for $UNDEFINE. |

Ordinary input lines are copied to the output. Directive lines are not copied. Lines within $COMMENT ... $ENDCOMMENT blocks are not copied. Lines within $IF ... $ENDIF blocks are either copied or discarded according to the expressions in $IF and $ELSEIF commands. Lines that begin with two dollar signs ($$) are copied (provided they are not within a block that should not be copied), but the first dollar sign is removed.

## A6.4. Restrictions and Other Details

COMMENT blocks can be nested to any depth.

Anything can appear in a COMMENT block, even if it looks like another pre-processor directive. In particular, $IF and similar commands inside COMMENT blocks are ignored, so a $COMMENT directive cannot be disabled by enclosing it in an IF block.

IF blocks can be nested up to a maximum depth which is determined when the Condpp program is compiled. This maximum depth is reported when the program is run with the **-P** option.

Expressions in IF blocks are restricted to the following forms:

|  |  |
|---|---|
| ***symbol*** | True if the named symbol exists. |
| **!** ***symbol*** | True if the symbol does not exist. |

Symbol names are not case-sensitive.

If a symbol definition is attempted when the symbol table is full, no warning is given. The maximum length of a symbol name and the maximum number of symbols that may be defined are determined when the Condpp program is compiled, and are reported when the program is run with the **-P** option.

It is not an error to undefine a symbol that was not defined.

Symbol names that are too long are truncated without warning.

Input lines that are too long are truncated without warning.

If a line that is to be copied to the output begins with two dollar signs ($$), the first dollar sign is removed. This allows the output to contain lines that would otherwise be interpreted as preprocessor directives.

Input lines that begin with a dollar sign ($) but are not recognised as pre-processor directives are treated like ordinary lines; it is unnecessary (but harmless) to double the initial dollar sign.

On some computers, Condpp can find the actual length of the input line, including any trailing blanks that might be present. Output lines will then be exactly the same as the corresponding input lines. On some computers, however, a Fortran program cannot find the actual length of an input line. In this case, all trailing blanks are removed from each output line.

## A6.5. Verbose or Debug Mode

In verbose (or debug) mode, which is enabled by the -V command line option, every input line is copied to the output, after having a six character prefix added to it. The last character of the prefix is a blank space if the corresponding line would normally have been output, and is a minus sign (−) if the line would not normally have been output. The first five characters of the prefix give information about the reason for a line's inclusion or exclusion. All possible prefix types are shown below. A dot (.) in the list below denotes a blank space in the prefix. *Nn* denotes a one- or two-digit number in the prefix.

| | |
|---|---|
| `......` | Ordinary line (blank prefix). |
| `$....-` | Preprocessor directive line. |
| `C.nn.-` | Line within a COMMENT block; *nn* is the COMMENT nesting depth. |
| `I.nn.-` | Line within an unsatisfied IF block; *nn* is the IF nesting depth. |
| `I.nn..` | Line within a satisfied IF block; *nn* is the IF nesting depth. |

## A6.6. Usage Examples

The following examples illustrate the effect of various command line arguments. They assume that the computer will run the Condpp program and pass it the corresponding arguments when a user issues the `condpp` command.

```
condpp -n -dabc mydefs - -o newfile
```

Cancel all predefined symbols, then define the symbol **abc**. Read input first from a file named **mydefs**, then from the standard input file. Send output to a file named **newfile**.

```
condpp definitions +b oldfile -o newfile
```

Start with the predefined symbols. Read input from the file named **definitions**. After processing the **definitions** file, turn on the blank lines flag (**+B** option) before taking further input from the file named **oldfile**. Send output to a file named **newfile**.

If **definitions** contains only preprocessor directives and comments, and does not generate any output lines, then corresponding lines in the output file (**newfile**) and the input file (**oldfile**) will have the same line numbers. This may be useful for relating compiler error messages generated from **newfile** back to the corresponding lines in **oldfile**. If the **+B** flag had been turned on before the processing of **definitions** then an extra blank line would have appeared in the output file for each line in the **definitions** file.

## Appendix A7. The Connect Program

This appendix contains detailed information about the Connect program. For the sake of completeness, much of the information presented in the body of the text is repeated here.

### A7.1. Introduction

The task of the Connect program is to interpret a specially prepared command file. Command line arguments passed to the program identify a terminal port on the computer and specify the name of the command file to be used. The command file instructs the program to perform simple tasks such as sending information to the terminal port, waiting for input data to be received, and making simple flow control decisions. The program also maintains a table of variable names (also referred to as *symbols*), with their definitions. All variables are character strings. Input received from the terminal port may be stored into a variable, variables may be output or used in comparisons, and the program's command line arguments may assign initial values to variables.

### A7.2. Command Line Arguments

The command line used to run the Connect program has the following format:

```
CONNECT commandfile [symbol=value] ...
```

The *commandfile* argument specifies the name of the command file to be interpreted by the program.

The square brackets [] and the ellipsis (...) above are not part of the command line, they simply indicate that the *symbol=value* part is optional and may be repeated any number of times. *Symbol* is the name of a variable that will be added to the program's symbol table, and *value* specifies the value to be assigned to the variable. A special syntax may be used to specify non-printable or other special characters in the *value* string.

Arguments are separated by space or comma characters. If the required symbol value contains space, comma, quotation mark or other special characters, the entire *symbol=value* string must be enclosed in single or double quotation marks ('...' or "..."). For example, if a user wishes to assign the value "**b c=d**" to the symbol named "**a**", he might gives the command

```
connect cmdfile "a=b c=d".
```

On some computers, quotation marks and some other characters may be used for special purposes by the command line interpreter, and may therefore not be present in the command line that is made available to a program. Special steps may need to be taken to ensure that the Connect program can see the quotation marks referred to above; any such action would of course be system dependent.

## A7.3. The Communications Line Used by the Connect Program

After processing any command line symbol assignments and opening the command file, the Connect program must open the serial communications port that will be used for the operations described by the command file. The name of the device that is used for this purpose can be specified in one of two ways. A symbol called **DEV** can be defined in the command line used to run the Connect program, or one or more **DEV** commands may appear at the start of the command file.

If the DEV symbol is defined in the command line used to run the program, then the program attempts to use the specified device, and ignores any DEV lines that may be present in the command file interpreted by the program. If the DEV symbol is not defined in the command line, then the program expects the command file to begin with one or more DEV lines. The device specified in the first DEV line is tried first. If it cannot be used, devices specified in subsequent DEV lines are tried. If the program cannot open any of the relevant devices, it terminates with an error message.

The device name given in the DEV symbol in the program's command line or in a DEV line in a command file is not necessarily the same as the name used by the operating system to refer to that device. This is because of the alias translation that may occur when the device is actually opened. Whether or not alias translations occur, the symbol named TRUEDEV will contain the name used by the operating system to identify the device.

## A7.4. Special Symbol Names

Some of the symbols that may be set in the command line used to run the Connect program, and may be examined or modified by commands in the command file, have special meanings to the program. Other symbols have no meaning other than that assigned by the writer of the command file. The following list describes all the symbols whose meaning is special.

**DEV**

This is the name of the device port on the local computer that will be used for communication with the network. If this symbol is set in the command line used to run the Connect program then any DEV lines in the command file are ignored. If the device name is in fact an alias, the DEV symbol retains the un-translated name, as it appears before aliasing occurred.

**TRUEDEV**

This is the true name of the device used for communication with the network. It is derived from the DEV symbol, but is affected by alias translations.

**OLDLOCK**

If this symbol is set, and its value is not simply a question mark (?), then lock method 4A (described in section A7.6) will be used when the device is opened. This means that the device must previously have been locked (by another program) and the *Lock Information* in the device lock file must match the value of this symbol.

If the symbol value is a question mark, lock method 4B is used. This will allow a locked device to be opened, provided the lock was not a strong lock. (A strong lock is one which has an exclamation mark as the first character of its lock information.)

The symbol must be set from the command line used to invoke the program if the action described above is desired, because its value is checked before any SET lines in the command file are executed.

Specifying the **OLDLOCK** keyword in a **DEV** command at the start of a command file automatically sets the **OLDLOCK** symbol, if the device specified by that **DEV** command is the device that is actually used.

**KEEPLOCK**

If this symbol is set when a simple **FINISHED** command (that does not specify the **KEEPLOCK** keyword) is executed, the device will be closed using lock method 3 (described in section A7.6). This will result in the device lock file containing the value of this symbol as the *Lock Information*.

**MAXSTEPS**

This symbol contains the maximum number of lines that may be interpreted before the program aborts. It should contain an integer number, expressed in ordinary decimal notation. Setting this symbol in the command line used to run the Connect program will have the desired effect, but setting it with a **SET** command in a command file will not work as expected.

The **MAXSTEPS** command, which should be used in a command file to change the maximum number of steps, does change the value of the **MAXSTEPS** symbol, but modifying the symbol value in any other way does not change the maximum number of steps.

**OUTPUT**

If this symbol is specified in the command line used to run the Connect program, then the program writes all its output to the file whose name is specified, instead of writing to the user's terminal. This does not affect output destined for the network connection.

## A7.5. Command Files

### A7.5.1. Introduction

The Connect program can use either printable or binary command files. Binary command files are prepared from their printable counterparts by the ConnToBin program, and are interpreted much more efficiently that are printable command files.

The Connect program searches in several places for the command file that it will use. On most systems, a file name may be divided into three parts: A directory part, that specifies where in the file system the file resides; a base name part; and an extension that may hint at the file's purpose or type. The Connect program checks for a file with no extension, with a .CCN extension, and with a .CON extension, in both the user's current directory and in a default directory in which standard Connect command files are stored. If a file has a .CCN extension, it is interpreted as a binary format command file. Binary command files are used in preference to printable command files with the same base name. The user can explicitly give a fully qualified file name (including a directory and an extension) to circumvent the program's searching strategy.

Each line in a (printable) command file used by the Connect program contains either a command or a comment. Blank lines and lines whose first non-blank character is a star (*), semicolon ( ; ) or exclamation mark ( ! ) are comment lines. Other lines contain commands. A comment may be appended to a command line if it is preceded by a semicolon ( ; ) or exclamation mark ( ! ). Commands may not cross line boundaries, and no more than one command may appear on a line.

Tokens that may appear in the command file include un-quoted words and quoted character strings. Character strings may be enclosed in either single or double quotation marks ( '...' or "..."). An un-quoted word may be used wherever a quoted character string is allowed, but the converse does not apply. Within a quoted character string, a quotation mark of the type used to delimit the string may appear twice in succession; this has the effect of inserting a single such quotation mark into the string. A special syntax (described in section A7.5.3) may be used to specify non-printable characters in any character string.

Any line may begin with a label. A label is an un-quoted word whose last character is a colon (:). The label may be used as the destination of a conditional or unconditional **GOTO** command. A line may have any number of labels, and a line with labels need not also include a command.

Wherever a character string is permitted, a symbol reference may be used. A symbol reference is an un-quoted word whose first character is a dollar sign ($). The name that appears after the dollar sign is the name of a symbol whose value is to be used. Symbol names are not case-sensitive. Symbols may be assigned values in the command line used to run the Connect program, as described in section A7.2, or by a **SET** command that appears in the command file. Some symbols are automatically set in various circumstances; these are described in section A7.4.

Each command line specifies a simple operation that must be performed. Whenever a command is executed, a step counter is incremented. When the step count exceeds a threshold determined by the **MAXSTEPS** command (or by the **MAXSTEPS** symbol, if it was set in the command line used to run the Connect program), the program will abort.

## A7.5.2. Commands in Connect Command Files

The following commands may appear in a command file for the Connect program. Text that must appear exactly as shown is in **boldface**. Words that will be replaced by suitable text are in *italics*. Square brackets [] represent optional items. Ellipsis (...) indicates that the previous item may be repeated any number of times.

**DEV** *device* [**OLDLOCK** [*oldlockinfo*]]

Specifies a *device* to be used for the connection. Lines of this form are ignored if they appear other than at the beginning of the file, or if a device was specified in the program's command line, or if a device has already been opened successfully.

Specifying the **OLDLOCK** keyword in a **DEV** command has exactly the same effect as defining a symbol named **OLDLOCK** in the command line used to run the Connect program: the device is opened provided it was previously locked and the previous lock information matches that specified.

If the **OLDLOCK** keyword appears without the *oldlockinfo* being given, the effect is the same as if **OLDLOCK=?** were specified in the command line used to run the Connect program: the device is opened if it was previously locked and the previous lock information does not begin with an exclamation mark, or if it was not previously locked.

**SEND** *string* ...

Send the specified character *string* to the network device. If more than one string is given, the component parts are concatenated without intervening blanks. An end-of-line character sequence is not automatically output; it must be specified explicitly if desired.

**RECEIVE** *symbol* [**SHOW**] **TIMEOUT** *secs* **GOTO** *label*

Read data from the device, until a timeout occurs or the end of a line is encountered. Store the input data into the *symbol* whose name is specified. If the **SHOW** keyword is specified, the received information will be displayed on the standard output device, with control characters encoded in a printable form, as for the

**SHOW** command. *Secs* specifies the timeout period in seconds. A fractional number of seconds may be specified, but the resolution and accuracy of the timeout period is system dependent. If a timeout occurs before a line of input has been received, any partial line of input will be stored into the relevant *symbol* and the program will transfer control to the given *label* in the command file. The timeout period applies to the delay between one character and the next, not to the total time taken for a line of input to be received.

A line of input ends with a carriage-return (CR) or line-feed (LF) character, but if a CR is followed immediately by an LF, or an LF is followed immediately by a CR, then both characters belong to the same line. A side effect of this approach is that, after receiving a CR or LF character, the program does not know whether that character was a line-terminator until either a timeout period expires or another character is received. To minimise the problems imposed by this strategy, the program shortens the timeout while reading the character just after either a CR or an LF. The shortened timeout period is 0.5 seconds, or as close to that value as the system permits.

**SHOW** *string* ...
**SAY** *string* ...

Display a character *string* on the standard output device, which is normally the user's terminal (not the device used for the connection). If more than one string is given, the component parts are concatenated without intervening blanks. The **SHOW** command displays control characters in a printable form, while the **SAY** command simply outputs any control characters that might be present, without modification. The codes used by the **SHOW** command to represent non-printable characters are the same as those used to specify non-printable characters in a character string in the command file. These codes are given in section A7.5.3.

**WAIT** *seconds*

Delay for the specified number of *seconds*. The delay period may include a decimal fraction, but the resolution and accuracy of the delay period is system dependent.

**FLUSH** [*seconds*]

Discard input in the system type-ahead buffers. If the optional number of seconds is specified, the program will attempt to ignore all input that arrives during that time. The implementation of this command is system dependent; it is guaranteed that all input in the system type-ahead buffers will be discarded, but input that arrives during the delay period may or may not be discarded.

**SET** *symbol string* **[FOLD]** [*string* **[FOLD]]** ...

Assign a value to the named *symbol*. The **FOLD** keyword specifies that the *string* must be converted to upper-case before being used. The word **FOLD** may be enclosed in quote marks to prevent this interpretation. If there are several character strings, they are concatenated without any intervening blanks, and each component may have its own optional **FOLD** keyword.

**ABORT**

Immediately abort processing of the script file. The original device lock information that may have existed before the program was executed is restored.

**FINISHED [KEEPLOCK** [*newlockinfo*]]

This command indicates normal termination of processing. If the **KEEPLOCK** keyword is specified without a value being given to *newlockinfo*, the effect is the same as if the symbol named **KEEPLOCK** has a question mark (?) as its value: the program will retain the device lock file without changing its contents. The device lock file will therefore contain the word **CONNECT** after the program terminates.

If the **KEEPLOCK** keyword is specified and a value is given to *newlockinfo*, the effect is the same as if the symbol named **KEEPLOCK** had been assigned the appropriate value: the program will not delete the device lock file, but will change the information in the lock file to reflect the *newlockinfo* specified.

**IF** *string1 operator string2* **GOTO** *label*

Compares the two character strings, and performs a transfer of control to the specified *label* if the test is satisfied. The *operator* may be one of the following:

| | |
|---|---|
| **=** | True if *string1* is identical to *string2*. |
| **< >** | True if *string1* is not identical to *string2*. |
| **BEGINS** | True if the beginning of *string1* is identical to *string2*. |
| **ENDS** | True if the end of *string1* is identical to *string2*. |
| **CONTAINS** | True if *string1* contains the characters of *string2*. |

GOTO *label*

Performs an unconditional transfer of control to the given *label* in the command file. In a printable command file, the GOTO command is implemented by rewinding the command file to the beginning, and searching for the destination label. This is very inefficient. In a binary command file, the exact destination of the GOTO command is stored, so that the transfer of control is much more efficient.

SYSTEM *string* ...

Perform an operating system command. If more than one character *string* is given, the components are concatenated (without intervening blanks). The resulting character string is executed as an operating system command.

### A7.5.3. Specification of Special Characters

The following character sequences are used by the Connect program to specify non-printable or other special characters. These special sequences may be used in Connect command files or in the command line used to run the program. Whenever one of these special sequences is encountered, the corresponding character is used instead. Upper- or lower-case versions of the characters in the sequences below are equivalent.

| | |
|---|---|
| `^`*C* | A control-*C* character. For example, `^@` generates an ASCII NUL character, char(0), and `^A` generates an SOH character, char(1). `^?` generates a DEL character, char(127). |
| `\r` | Carriage-return, char(13). |
| `\n` | Newline, or line-feed, char(10). |
| `\t` | Tab, char(9). |
| `\b` | Backspace, char(8). |
| `\e` | Escape, char(27). |
| `\f` | Form-feed, char(12). |
| `\s` | Blank space, char(32). |
| `\c` | Comma (`,`), char(33). |
| `\l` | End of line. This usually translates to a carriage-return and line-feed pair (equivalent to `\r\n`), but is system dependent. |
| `\z` | This translates to nothing at all. Sometimes useful for specifying a completely empty (zero-length) character string. |
| `\`*nnn* | Char(*nnn*), where *nnn* is an octal number of 3 digits or fewer. |
| `\o`*nnn* | Char(*nnn*), where *nnn* is an octal number of 3 digits or fewer. (The character after the backslash is a letter O.) |
| `\d`*nnn* | Char(*nnn*), where *nnn* is a decimal number of 3 digits or fewer. |
| `\x`*nn* | Char(*nn*), where *nn* is a hexadecimal number of 2 digits or fewer. |

| | |
|---|---|
| \\ | Generates a single backslash character (\). |
| \^ | Generates a circumflex character (^). |

### A7.5.4. **Conversion of Command Files to Binary Format**

The Connect program can use either a printable or a binary form of command file. Binary command files are interpreted more efficiently than are printable command files.

The ConnToBin program converts a printable command file to a binary command file. It may be run with up to two command line arguments. The first command line argument is the name of the printable command file to be converted, and the second argument is the name of the binary command file to be created.

If no command line arguments are given, the program prompts the user to enter suitable file names. If only the first argument is given, the program automatically chooses a suitable name for the binary output file.

The binary command file created by ConnToBin is accessed by the Connect program as a Fortran unformatted, direct access file, with a record length of 256 characters.

Each keyword is encoded as a single byte in the binary file. Ordinary words that are not keywords are encoded as a special flag byte, followed by a byte representing the length of the word, and followed by the characters that make up the word. Quoted character strings and references to symbols are encoded in the same way as ordinary words, but the flag byte is different in each case. The destination for a GOTO operation is encoded as a special flag byte, followed by three bytes which represent a 16-bit record number within the command file and an 8-bit offset within that record. The end of each line and the end of the command file are each encoded as a single byte.

As the ConnToBin program reads and converts the input file, it builds a table that defines the actual address for each label defined in the command file. Whenever a **GOTO** or conditional **GOTO** command appears, its position within the binary file is noted in another table. When the entire input file has been read and converted, the contents of the two tables are used to fix up the **GOTO** destinations in the binary file.

## A7.6. **Device Locking**

Because of the large differences between various operating systems' treatment of true device locking, the Connect program uses a different method to decide whether it is permitted to use a given device. Implementations of the program on some systems may also use operating system locks, but the basic method used by the Connect program relies on cooperation between programs attempting to use a given device.

A system of lock files is used for this purpose. There is a *Device Lock File* for each device that may be used by the Connect program; the existence of one of these lock files means that the program that created the lock file is permitted to use the associated device. Another lock file, called the *Exclusion File*, gives the program that created it permission to create, modify or delete device lock files.

## A7.6.1. **The Device Information File**

A system-wide *Device Information File* (referred to simply as the *INFO* file) lists the devices that the Connect program is permitted to use for network connections. Each device has a *Device Lock File* associated with it, and the INFO file specifies the name of the Device Lock File for each device. If the lock file associated with a particular device exists, cooperating programs (including the Connect program) will not attempt to use the device.

If the lock file associated with a device that the Connect program wants to use does not exist, then the program creates the lock file, thus securing exclusive access to the device. The presence of a device lock file, of course, does not prevent another program (which might be either malicious or merely ignorant of this locking mechanism) from accessing the device; Locks provided by the operating system are the only variety that can guarantee such protection.

There are three types of lines in the INFO file: *Comment* lines, *Alias Translation* lines and *Lock Information* lines. Blank lines or lines whose first character is a star (*) are treated as comment lines, and are ignored. Alias translation lines have the form

*oldname=newname.*

Blank spaces are not permitted in alias translation lines. Lock information lines have the form

*devicename lockfilename.*

The DeviceLock subroutine, which handles device locking, scans the INFO file sequentially. If an alias translation line is found whose *oldname* corresponds to the current device name, then the current device name is replaced by the *newname* specified in the alias translation line. This action may occur several times, if the new name specified in one alias translation matches the old name specified in another alias translation, provided the lines are in a suitable order. When a lock information line is found whose *devicename* corresponds to the current device name, then the name of the Device Lock File is determined from the lock information line. Distinctions between upper-case and lower-case device names are ignored for comparison purposes.

The DEV symbol used to identify the device used for the connection is unaffected by alias translations. The name that appears in the lock information line that is used (after all alias translations have taken place) is eventually saved in the **TRUEDEV** symbol, in exactly the same form (upper- or lower-case) as it appears in the INFO file.

### A7.6.2. The Exclusion File

After determining the name of the Device Lock File to be used, the program must create the Device Lock File in order to lock the device. Since several programs might simultaneously attempt to lock the same device, some way of ensuring that only one program at a time can create lock files is required. The method that is used depends on an additional lock file, called the *Exclusion File*. If the Exclusion File exists, only the program that created it is allowed to create, delete or modify Device Lock Files.

The method of maintaining the Exclusion File is described in sections A7.6.6 and A7.6.7. This method is believed to be safe from deadlocks provided two conditions are met: Firstly, if several simultaneous requests to create, open, or delete a file are made, the operating system must behave in one of the ways mentioned in section A7.6.6.4 as being acceptable; Secondly, the timing considerations described in sections A7.6.6.1 and A7.6.6.4. must be satisfied.

The DeviceLock subroutine, which handles the device lock files, makes use of two system-dependent subroutines, called MakeLockFile and KillLockFile, to control access to the Exclusion File. The MakeLockFile subroutine is used to create the Exclusion File. This is done before a Device Lock File is accessed. The KillLockFile subroutine deletes the Exclusion File after access to the Device Lock File is completed.

### A7.6.3. The Device Lock File

The first line in the Device Lock File contains a character string known as the *Lock Information* or the *Lock Reason*. The lock information will usually be a brief indication of the reason for the lock. It is this information that is set when a device is closed with the KEEPLOCK option or opened with the OLDLOCK option.

Subsequent lines in the device lock file contain the date and time at which the lock was created, the name of the user who created the lock, and the identity of the user's terminal.

### A7.6.4. Actions Supported by the DeviceLock Subroutine

The Connect program uses a subroutine named **DeviceLock** to handle the creation, modification and deletion of device lock files. This subroutine also performs device name alias translation. The DeviceLock subroutine accepts six arguments, as follows:

```
      subroutine DeviceLock (DeviceName,Action,
     +                   OldInfo,NewInfo,PrevInfo,Error)
      character*(*) DeviceName,OldInfo,NewInfo,PrevInfo
      integer Action,Error
```

*DeviceName* is the name of the device to be locked. This character string will be modified when the subroutine returns, if the device name is found to be an *alias*, as described in section A7.6.1. *Action* specifies what the subroutine must do: values between 1 and 4 cause the operations described below. *OldInfo* specifies the lock information that must already be present in the device lock file, if the action code is 4. *NewInfo* specifies the new lock information, if the action code is 1 or 3. *PrevInfo* is an output parameter; it contains the lock information that was previously in the device lock file, if the action code was 1 or 4. *Error* is an output parameter that will be non-zero if the subroutine fails to perform its task.

Depending on the options passed to the DeviceLock subroutine, it may create a new lock file containing specific lock information, check that the existing lock information is correct, modify the lock information, or delete the lock file entirely. The following are the actions that may be performed by the DeviceLock subroutine:

Action code 1

The device is locked if it was not already locked. This is the normal action before a device is opened by the Connect program. The device lock file is created with *NewInfo* as the lock reason.

Action code 2

The device is unlocked, provided it was in fact locked. This is the normal action when a device is closed by the Connect program. The device lock file is simply deleted.

Action code 3

The information in the device lock file is unconditionally changed, provided the lock file exists. This action is performed when a program wishes to leave a device locked after the program has terminated. *NewInfo* would normally be chosen in such a way as to allow a subsequent program to open the device using method 4A.

Action code 4, case A

This operation is performed when the action code is 4 and *OldInfo* is not a question mark (?). The device is locked, and *NewInfo* is made the Lock Reason, only if both the following conditions hold:

a) The device is already locked.

b) The Lock Information in the device lock file matches the expected information supplied in the *OldInfo* parameter. (This comparison does not differentiate between upper- and lower-case letters.)

This is intended for use when several programs must use a device in sequence. The first program will lock the device and leave a special value in the device lock file that allows the second program to use it; subsequent programs will set lock values allowing their successors to use the device; the last program in the sequence will unlock the device.

Action code 4, case B

This operation is performed when the action code is 4 and *OldInfo* is just a question mark (?). The device is locked, and *NewInfo* is made the Lock Reason, if either of the following conditions holds:

a) The device is not already locked.

b) The device is already locked, and the Lock Information does not begin with an exclamation mark (!).

This allows an *weak lock* to be disregarded, but a *strong lock* (whose information begins with an exclamation mark) cannot be broken by this method. This action is intended for error recovery.

### A7.6.5. Use of Device Locks by the Connect Program

This section describes how the Connect program manipulates Device Lock Files in various situations. The DeviceLock subroutine is used for all Device Lock File manipulation. The lock method numbers used below refer to the descriptions in section A7.6.4.

Normal device open:

Unless directed otherwise by options in the command line or the command file, the Connect program ensures that the device is unlocked and then locks the device before using it. (DeviceLock action 1.) The lock information placed in the device lock file is the word **CONNECT**.

Device open with previous lock:

If the command line sets the symbol **OLDLOCK** or if the relevant **DEV** line in the command file specifies the **OLDLOCK** keyword, the Connect program will call the DeviceLock subroutine with action code 4. One of two possible actions will then occur.

If the value of the OLDLOCK symbol (or the value associated with the OLDLOCK keyword) is anything other than a question mark (?), the program will lock the device only if it is already locked and the existing lock information is identical to the specified value (without upper-case and lower-case characters being differentiated). (This is DeviceLock action 4A.)

If the OLDLOCK symbol is a question mark, or if the OLDLOCK keyword has no value associated with it, the Connect program will lock the device either if it is not locked or if it is locked with a weak lock (that is, a lock whose information does not begin with an exclamation mark). (DeviceLock action 4B.)

Normal device close:

The Connect program will usually unlock the connection device just before terminating. (DeviceLock action 2.)

Close device and retain lock:

If the symbol **KEEPLOCK** is defined, and is not just a question mark (?), when the Connect program terminates normally, or if the **FINISHED** line in the command file specifies the keyword **KEEPLOCK** with its associated value, then the program will leave the device locked. The KEEPLOCK symbol specifies the lock information to be placed into the device lock file under these circumstances. (DeviceLock action 3.)

If the KEEPLOCK symbol contains just a question mark, or if the KEEPLOCK keyword is specified without a value in a FINISHED line in the command file, then the device lock information will retain the value it has while the Connect program is running. This will be the word **CONNECT**. (The program does not call the DeviceLock subroutine at all in this case.)

Abnormal termination:

If the Connect program terminates by exceeding the maximum number of command lines interpreted, by attempting to execute an invalid command line, or by executing an **ABORT** command in a command file, the device lock information will be restored to the value it had before the program started. This involves either deleting the device lock file (DeviceLock action 2) or changing the lock information (DeviceLock action 3).

If the operating system aborts the program, the results are unpredictable. Usually, the device lock file will be left with the word **CONNECT** as the lock information.

### A7.6.6. **The MakeLockFile Subroutine**

The purpose of the MakeLockFile subroutine is to create a short-lived *Lock File*. When this subroutine is called by the DeviceLock subroutine, the Lock File that is created is the Exclusion File described in section A7.6.2. Because Lock Files may be deleted automatically when they reach a certain age, this subroutine is not suitable for managing long-lived lock files.

If the Lock File already exists, MakeLockFile will usually delay a short time before trying again to create the Lock File, and will loop in this way until the Lock File has been created or a severe error is encountered. It is possible to prevent this retry loop by passing suitable arguments to the subroutine, but the DeviceLock subroutine does not do so.

A7.6.6.1. **Lock File Obsolescence**

To guard against deadlocks caused by a program terminating without deleting the Lock File, MakeLockFile assumes that an already existing Lock File older than a certain number of seconds is obsolete. An obsolete Lock File is simply deleted, on the assumption that the program that created it has crashed for some reason.

If the program that created the Lock File is still busy with the operations for which it created the Lock File, when another program notices that the lock file has become obsolete, then each programs will believe that it holds a valid exclusive lock. This would make the Lock File useless.

The number of seconds chosen as the obsolescence age of the Lock File is therefore critical. This period must allow the creator of the Lock File time to perform whatever protected operations are required, but must not be long enough to cause undue delay if a program does abort without deleting the Lock File.

The DeviceLock subroutine does not do much work while the Exclusion File exists: It performs not more than twelve file system operations (create, open, read, write, close or delete), depending on the particular options in effect. These operations are completed in under 2 seconds on the HP-A900 computer, when lightly loaded, and the HP-9000 and VAX-750 are slightly faster.

A period of 20 seconds was chosen as the obsolescence age for the Exclusion File. This allows sufficient time to cope with delays in a fairly heavily loaded system, while still allowing rapid recovery from deadlocks caused by failure to delete the Exclusion File.

A7.6.6.2. **Arguments Passed to MakeLockFile**

The MakeLockFile subroutine accepts three arguments, as follows:

```
subroutine MakeLockFile (LockFile,Age,Error)
character*(*) LockFile
integer Age,Error
```

*LockFile* is the name of the Lock File to be created. *Error* is an output parameter that will be non-zero if the subroutine fails to perform its task. *Age* is the obsolescence age of an old lock file, with the following special meanings:

> If Age is positive, the subroutine must loop until the Lock File is created successfully and must treat an old Lock File older then the specified number of seconds as obsolete.

If Age is zero, the subroutine must neither loop nor delete an old Lock File, it must simply return an error code if an old Lock File already exists.

If Age is negative, the subroutine must not loop, but must delete an obsolete Lock File, if one is found.

## A7.6.6.3. **Algorithm for the MakeLockFile Subroutine**

The following algorithm describes the action of the MakeLockFile subroutine. This subroutine has been implemented, substantially in this form, on all the computers that support the Connect program. The lock file is never opened for exclusive access (if the operating system supports exclusive access to a file); at any time, any number of programs may have the same lock file open. The steps labelled A, B, C, D and E are discussed in section A7.6.6.4.

```
 loop forever /* until a RETURN from the subroutine */
A:  try to create the lock file
    if created successfully then
      assign permission for anybody to delete the lock file
      close lock file
      return with error=0
    else /* could not create */
      if reason for failure is not that file already exists then
        return error indication /* something serious is wrong */
      endif
      if age is zero, return error indication
                  /* must not loop if age=0 */
B:    try to open lock file as an existing file
      if failed to open then
        if reason for failure is file does not exist then
          /* another process has probably just deleted it */
          /* loop and try again */
        else /* if some other reason for failure */
          return with error<>0
        endif
      else /* managed to open existing lock file */
        get file creation time
        calculate file age
        if file is too old then
          /* another process probably died without cleaning up */
C:        try to close and delete the lock file
```

```
            if failed to delete then
              if reason for failure is file does not exist then
                /* another program probably just deleted it */
                close the lock file
                /* loop and try again */
              elseif failure is because another program
                    has the file open, then
                sleep about 1 second
                /* to give another program a chance */
                /* to delete the file */
              else /* some other reason for failure */
                close the lock file
                return error indication
              endif
            else /* managed to delete the obsolete lock file */
D:              sleep about 1 second /* to avoid race conditions */
            endif
          else /* the old lock is not obsolete */
            close the lock file
            if age is negative then return with error< >0
          endif
        endif
      endif
      if age is negative, return with error indication /* don't loop */
E:    sleep a short time /* give lock owner time to release it */
    endloop /* loop and try again */
```

A7.6.6.4. Possible Outcomes of Various Stages in MakeLockFile

Some possible situations that may arise at various stages in the MakeLockFile algorithm are discusses in the following sub-sections. References to steps A through E apply to the labelled steps in the algorithm given in section A7.6.6.3.

A7.6.6.4.1. Creating the Lock File

There are several conceivable outcomes to the attempted creation of the lock file (step A). These are discussed for the cases of only one program or of several programs simultaneously attempting to create the same lock file.

If there is only one program, and a lock file already exists, the file system must (a) report to the program that the creation failed because the file already exists. If the file does not already exist, the file system must either (b) create the file and report that it has done so, or (c) report that some other error occurred (for example, the user might not have permission to create the file).

If more than one program simultaneously attempts to create the lock file, and the lock file already exists, then all the attempts must fail, as in case (a) above. If the lock file does not already exist, some of the attempts may fail for reason (c) above. There are also several other possible outcomes: (d) All the attempts may fail, without the lock file being created; (e) The lock file may be created, but the file system reports that all the attempts failed; (f) The file may be created and exactly one of the competing programs is informed of its success, while all the other attempts are failed by the file system; (g) The file may be created and more than one program informed that it was successful; (h) The file may not be created, but one or more programs is informed that it was successful.

Of all these possible outcomes, (a), (b), (c) and (f) are completely harmless. A well-designed file system should always react in one of these ways. In case (a), each program simply loops, trying again until either it succeeds or the lock file becomes obsolete. In case (b), the successful program can proceed immediately, having achieved its objective. In case (c), the unsuccessful program is immediately informed of failure by the MakeLockFile subroutine. In case (f), the successful program can continue, as for case (b), while the unsuccessful programs loop, as in case (a).

Cases (d) and (e), while not ideal, are also not very dangerous. In case (d), all the programs simply loop, until one of them is eventually successful. In case (e), all the programs loop until the spuriously created lock file becomes obsolete. Clearly, case (e) is worse than case (d) in respect of the impact on performance.

In cases (d) or (e), it is possible for the conflict to repeat indefinitely, with the same result each time the programs try to create the file. If the programs delay for a random period (instead of a fixed period) each time they execute the retry loop, this possibility will be eliminated.

Cases (g) and (h) are dangerous, because they can lead to several programs each believing that it holds an exclusive lock. The way this occurs in case (g) needs no further explanation. In case (h), another program can successfully create a lock file while the program that was misinformed by the file system still believes itself to hold the exclusive lock.

File systems that exhibit behaviour giving rise to cases (g) or (h) above, or giving rise to outcomes not considered above, are entirely unsuitable for this purpose. File systems in which cases (d) or (e) can occur are usable, but undesirable. File systems in which only cases (a), (b), (c) and (f) can occur are the most desirable.

### A7.6.6.4.2. Opening an Existing Lock File

When a program fails to create a new lock file, it attempts to open the lock file as an old, or already existing, file (step B above). If (a) it is successful, then the subroutine simply goes on to determine whether the lock file is obsolete. There are two possible reasons why opening the existing lock file might fail: (b) The file might no longer exist; or (c) some other error might occur.

In case (b), some other program must have deleted the file after the unsuccessful program failed to create it (at step A), but before it tried to open it (at step B). This is no problem, the unsuccessful program simply loops and tries again.

In case (c), the unsuccessful program must return an error code because it is unlikely that simply trying again will solve whatever problem is responsible for the failure.

### A7.6.6.4.3. Deleting an Obsolete Lock File

When an obsolete lock file is encountered, there are several possible outcomes to the attempted deletion of the lock file (step C above).

If only one program attempts to delete the obsolete lock file, then either (a) the file should be deleted successfully, or (b) the file system should report an error (for example, the user may not have permission to delete the file).

If several programs simultaneously, or almost simultaneously, attempt to delete an obsolete lock file, one of the following possibilities could occur: (c) One of the deletion attempts is successful, and the file system reports that all the other attempts were unsuccessful; (d) All the attempts fail because the program trying to delete the file is not the only program that has it open; (e) More than one of the deletion attempts is reported as successful; (f) The file is deleted, but all the programs are informed that they were unsuccessful; (g) One or more of the attempts is failed for some reason other than those in cases (c), (d) or (f).

Cases (a), (c) and (d) are the most desirable. In case (a), the program will loop and try again to create the lock file, while in case (c) both the successful and the unsuccessful programs will loop. Case (c) is likely to occur if the file system requires programs to delete files by name, while case (d) is likely to occur if the file system requires files to be closed and deleted in a single operation. In case (d), the unsuccessful programs close the file and delay for a short period. The last of the competing programs should then be able to delete the file successfully, but it is possible for the conflict to repeat indefinitely as all the programs loop again. Delaying for a random period instead of a fixed period would solve this problem.

Case (b) will occur only if the program does not have permission to delete the lock file; this should never occur because whenever a lock file is created, the creator sets its protection in such a way that any other user or program can delete it. Case (e) is not ideal, but is harmless; all the programs involved will simply loop and try again, in much the same way as for case (c). Case (f) is also harmless; all the programs will loop, discover that the file has been deleted, and loop again to attempt to create a new lock file. Case (g) is very undesirable because it would lead to the program concerned failing to secure a lock; file systems in which this could occur are unsuitable for this purpose.

#### A7.6.6.4.4. Delay After Deleting an Old Lock File

The following scenario provides an explanation of the need for the delay at step D. Two programs may decide that a particular lock file is obsolete, and may then both try to delete it. The first of these two programs may successfully delete the lock file and continue on its way, after the second program has established that the file is obsolete but before the second program actually tries to delete the file.

If the first program immediately creates a new lock file, it is possible that the second program will delete this newly created file instead of deleting the obsolete lock file (which will already have been deleted by the first program). If this occurs, the first program believes that it holds the lock, but the second program believes that there is no lock. The second program will then proceed to make itself another new lock file. Now each program believes itself to hold an exclusive lock, and this is clearly unacceptable.

The basic problem in the above scenario is that the first program was too quick to create a new lock file after deleting the obsolete lock file. This problem is not likely to arise unless the file system requires files to be deleted by name. If the file system allows an open file to be closed and deleted simultaneously, it is likely that either the first program would fail to delete the file, because it is open to the second program, or the second program should fail to delete the file, because it has already been deleted by the first program although a new file with the same name now exists.

The additional delay at step D in the algorithm, together with the normal end-of-loop delay at step E, guards against the problem described here. The total delay time (at steps D and E) must allow the second program (in the above scenario) time to get from step B (opening the old lock file) to step C (attempting to delete the obsolete lock file) before the first program can loop back from step C to step A and create a new lock file. In exceptional circumstances, the delay time that is actually used by the subroutine might be insufficient, and there would be no way for the programs involved to know that the algorithm had failed.

### A7.6.7. **The KillLockFile Subroutine**

The KillLockFile subroutine simply deletes the lock file created by the MakeLockFile subroutine. It has two arguments, the name of the lock file and an error return argument. This subroutine does not loop, it simply makes one attempt to delete the lock file and returns a non-zero error code if it fails.

The following algorithm describes the operation of the KillLockFile subroutine. This subroutine has been implemented, substantially in this form, on all the computers that support the Connect program. The algorithm is sufficiently simple and self-explanatory to need no further comment.

```
try to open existing lock file
if failed to open then
   return with error< >0
else /* managed to open lock file */
   try to delete lock file
   if failed to delete then
      close lock file
      return with error< >0
   else /* managed to delete lock file */
      return with error=0
   endif
endif
```

## A7.7. **Log File Used by the Connect Program**

The Connect program appends an entry to a log file as it starts executing, and appends another entry when it terminates. If the log file does not exist, or if it cannot be opened, no error is reported and the data logging is simply not performed. The name of the log file is system dependent.

The log file is protected against simultaneous access by several programs, using a lock file. The AppendLog subroutine that writes to the lock file uses the MakeLockFile and KiliLockFile subroutines to handle creation and removal of the lock file.

The entry added to the log file includes the user's name and terminal, the date and time, the code name or number used by the operating system to identify the program, and a description of the event being logged. At the start of the program, the command line arguments passed to the program are included in the log file. If the program terminates due to an error, the error message is included in the log.

Ordinary Fortran file operations are used to access the log file. The entire sequential file is read by the program, until an *end of file* error occurs, after which the log data is written to the file. This method is slow, taking several seconds when the log file has grown to several hundred lines, but standard Fortran does not provide any other way of appending data to a sequential file.

The log file should be shortened at regular intervals to avoid long delays in this subroutine. The log file may also be deleted entirely, in which case it will not be re-created by the Connect program.

## A7.8. Error Reporting by the Connect Program

When the program is successful (and executes a **FINISHED** command in a command file), it terminates without printing any special messages. If it is unsuccessful, it prints a message indicating the program's version number and revision date, and the nature of the problem. If the problem has to do with a command file, the current position within the file is also reported. The entry appended to the log file at the end of the program also contains the text of the error message.

Most operating systems provide a way for a program to return an exit status code when it terminates. This status code would then be available to the parent program or command file that invoked the newly terminated program. The Connect program will exit with one of three possible status codes, signifying normal completion, failure due to performing too many steps, or failure due to a severe error.

## A7.9. System Dependencies in the Connect Program

Because of the differences between the various computer systems on which the Connect program has been implemented, there are some system dependencies in the program. These system dependencies are most apparent in the parts of the program that deal with input/output (to both the communications device and the user's terminal) and those that deal with lock files. The method of installing the program so that it can be used by any user, the way command line arguments are passed to the program, and the file names used for various special purposes are also system dependent.

Many of the system dependencies are masked by the use of the Condpp preprocessor, so the same master source code can be transported between the three supported systems (the HP-9000, HP-A900 and VAX-750), and compiled without any manual modifications.

Various low-level routines are not included with the master source code, but are supplied on each system. The subroutines in this category are those used for device I/O, lock file management, command line argument access and termination with a status code.

The following sub-sections describe the implementation of system dependent features on all three computers.

### A7.9.1. HP-9000 Device I/O

On the HP-9000 computer, the low-level subroutines that handle I/O to the connection device are written in C, using standard Unix system and library calls. At the start of the program, the device is put into a mode in which the terminal driver performs flow control but does not do any other special processing (no echo, no line editing, no special interrupt characters). The flow control method is not explicitly set up when the device is opened, so XON/XOFF flow control will be done only if it was enabled before the Connect program was started. The original configuration is restored at the end of the program. The system's terminal device driver handles type-ahead buffering, as usual.

When the device is opened, the program checks that it is a terminal and that the current user is allowed to read from and write to the device.

Input with timeout is implemented using timer alarms. Unix allows only one alarm event to be pending (for a particular process) at any one time, so it is necessary to save the old alarm handler address and the old alarm time before setting up the new alarm handler and time. The old alarm handler and alarm time are restored (as accurately as possible) after the input character has been read. Input is read one character at a time, until the software determines that an entire line has been read.

The timeout period (as well as the period for the WAIT command) has a resolution of 1 second on the HP-9000 computer. This restriction is imposed by the system alarm() and sleep() functions.

A FLUSH command that includes the optional delay period will work exactly as described in section A7.5. The terminal device driver's CREAD flag is turned off for the specified period, so all input during that period is discarded [HP-UX, section term(4)].

The operating system does not provide a method of locking files or devices for the exclusive use of a particular program. Some standard software, such as the UUCP networking system, expects to find a lock file in a particular directory if a device is "locked" to some program. The Connect program can easily fit in with this scheme, if the names of the Device Lock Files (given in the INFO file) correspond to those needed by the other software. We do not use any software that requires this scheme to be used, so no attempt has been made to make the Device Lock File names fit in with any externally imposed standard.

### A7.9.2. **HP-9000 Terminal I/O**

All I/O to the user's terminal is done using Fortran READ and WRITE statements. The non-standard "$" format descriptor is sometimes used to suppress the end-of-line characters usually sent to a terminal by a Fortran WRITE statement.

### A7.9.3. **HP-9000 Lock Files**

The MakeLockFile and KillLockFile subroutines on the HP-9000 are written in C, using standard Unix library functions.

Files must be deleted by name, and it is possible for one program to delete a file that another program is using. What happens in this event is that the file remains available to the second program, but its directory entry is removed and it therefore cannot be opened by any more programs. Another file with the same name can now be created. The storage space associated with a file that is no longer catalogued in a directory remains valid until the file is closed by any programs that had it open when the deletion took place. This means that the problems described in sections A7.6.6.4.3 and A7.6.6.4.4 can occur if the delay used at step D in the MakeLockFile algorithm is too short.

### A7.9.4. HP-9000 Installation and Command Line Arguments

To install the program so that it can be used by any user involves copying the executable file to one of the standard directories that are automatically searched whenever a user types a command. It is also necessary to set suitable access permissions on the executable file. The Connect program resides in the **/usr/local/bin** directory, and has access permissions that allow any user to run the program. In keeping with the Unix convention of using lower-case names for most files, the Connect program is actually named "**connect**" (spelled using only lower-case letters) on the HP-9000.

When the Unix operating system starts a program, the program is passed an array of character strings comprising its command line arguments. If the main program were written in C, it would be declared as

```
main (argc, argv)
int     argc;  /* the number of elements in the argv[] array */
char  * argv[]; /* an array of character strings */
{
   /* the program body goes here */
}
```

**argv[0]** is a null-terminated character string that, by convention, contains the program's name. **argv[1]** to **argv[argc-1]** are null-terminated character strings that contain the program's command line arguments. Note that there could be an arbitrary number of these arguments.

When a program is written in Fortran, the user-written main program is not executed immediately. Instead, a part of the Fortran run-time library is executed, and this transfers control to the user-written main program. The **argv** array described above is not available to the user-written main program. Instead, a non-standard form of the PROGRAM statement can be used to make command line arguments available to the program. For example, a program declared as

```
      PROGRAM MAIN (A,B,C)
      CHARACTER*(*) A,B,C
```

would have access to the first three command line arguments, with the variable **A** corresponding to **argv[1]**, **B** to **argv[2]**, etc. While this method does allow any number of arguments to be passed to a program, it requires the number of arguments to be known when the program is compiled.

Because it is not possible to know at compile-time how many arguments will be passed to the Connect program, some other method of accessing the command line arguments is required.

Fortunately, the compiler-supplied startup code saves the argc and argv parameters in globally accessible variables named **_argc** and **_argv**. A subroutine named **getrunstring**, written in C, retrieves the command line arguments using these global variables. The arguments are returned in a Fortran CHARACTER variable. The **getrunstring** subroutine concatenates all the strings from the **argv** array into a single string, with a blank space separating adjacent elements. Any arguments from the **argv** array that contain embedded blank spaces, commas, or quotation marks, are enclosed in quotation marks by the **getrunstring** subroutine. This allows the main program to process the command line arguments correctly.

If a user wishes to set the symbols **dev**, **a** and **c** to particular values when he runs the Connect program, he might type the following command to the standard Unix command line interpreter (the Bourne Shell):

```
connect commandfile "dev=tty07" a=b 'c=d " e f'
```

The **argv** array will then contain the following character strings (the colons delimiting the start and end of the strings do not form part of the strings):

```
argv[0] = :connect:
argv[1] = :commandfile:
argv[2] = :dev=tty07:
argv[3] = :a=b:
argv[4] = :c=d " e f:
```

Notice that the Shell has broken the command line into several *arguments* and that the quotation marks around the second and last arguments have been removed. The **getrunstring** subroutine will convert this into a single Fortran character string, as follows (again, the colons are not part of the string):

```
runstring = :commandfile dev=tty07 a=b "c=d "" e f":
```

Notice that the section corresponding to **argv[4]** has been enclosed in quotation marks and that the quotation mark inside that section has been doubled. The quotation marks that originally enclosed the second argument were not necessary, and have been lost. The HandleRunstring function in the main part of the Connect program will now set the following symbol values:

```
dev = :tty07:
a   = :b:
c   = :d " e f:
```

These symbol values are identical to those that would have been set if the getrunstring subroutine had returned the command line exactly as it was typed by the user, namely

```
runstring = :commandfile "dev=tty07" a=b 'c=d " e f':
```

The operations described above do have the desired effect, although the method is somewhat clumsy. It might be preferable for the main part of the Connect program (specifically the HandleRunstring function) to have more knowledge of the Unix command line argument passing method, instead of insisting that the getrunstring subroutine return a single character string. The use of the getrunstring subroutine is necessary on other computers however, and using the method described here makes the HandleRunstring function itself entirely system-independent, requiring only that a suitable getrunstring subroutine be provided on each computer.

### A7.9.5. **HP-9000 Exit Status**

The program terminates by executing an *exit* system call, with an argument value of 0 (for success), 1 (for too many steps) or 2 (for ABORT or other failure).

### A7.9.6. **HP-9000 File Names**

The *Device Information File*, or *INFO* file, is named **/usr/conn/device/INFO**. The log file is named **/usr/conn/device/LOGFILE**. The standard directory for command files is **/usr/conn/scripts**.

### A7.9.7. **HP-A900 Device I/O**

Under the RTE-A operating system, there is no device-independent way of setting device parameters such as input timeout period, and input echo or no-echo mode. Another problem is that, when a program uses an ordinary WRITE statement followed by a READ statement (to output information to a device and read the device's response), some of the input data is often lost because the interface does not change from output- to input-mode sufficiently rapidly and the standard device drivers do not provide input type-ahead buffering.

Hewlett-Packard products usually avoid the input data loss problem by using a special handshake protocol. One of the simplest form of this protocol (which has several variations) requires the computer's interface to output an ASCII *DC1* character (also called an *XON* or *control-Q* character) when it is ready for input. The external device may not send data without having received the *DC1* character, and may not send too much data (more than about 80 characters) in a single burst. Some device interfaces also support other handshake protocols, and the DC1 trigger handshake can usually be disabled. On most of the ports connected to EENET, *XON/XOFF* protocol is used. Although some of the HP interfaces claim to support this protocol, they do so in one direction only: an external device can send an *XOFF* to tell the computer to stop sending data, but the computer will never send an *XOFF* to tell the device to stop.

On the HP-A900, it is necessary for a program (such as the Connect program) that requires low-level control of a device, to use device dependent system calls. A program can use *EXEC* calls to perform various functions, including device I/O. **EXEC 1**, **EXEC 2** and **EXEC 3** calls can be used to read, write and control any type of device, but the exact results depend on the type of device and on the system *Device Drivers* and *Interface Drivers* in use. Certain control bits used in read and write requests are device dependent, and most I/O control requests are device dependent.

There is also another type of I/O called *CLASS I/O*. This allows I/O to proceed asynchronously, leaving the program free to perform other tasks while the I/O transfer occurs. To use Class I/O, a program must be allocated a *Class Number*, which it uses in *Class Read*, *Class Write*, *Class I/O Control* and *Class Get* calls (EXEC 17, EXEC 18, EXEC 19 and EXEC 21).

Each device has a queue of pending I/O requests. The Class Read, Class Write and Class I/O Control calls add read, write and control requests (similar to the normal EXEC 1, EXEC 2 and EXEC 3 read, write and control requests) to the device's queue. Each class number has a queue to which completed requests are transferred. The Class Get call retrieves completed requests from this queue. Various control bits in the Class Get call allow the program to return immediately if no data is available, or to wait until data is available. The system **CLRQ** subroutine can be used to allocate and de-allocate Class Numbers and to flush pending Class I/O requests.

To receive a high-speed burst of data, a program can use a technique called *Double-Buffered Class I/O*, or the more general *Multiple-Buffered Class I/O*. The following algorithm illustrates the technique:

```
Allocate a Class Number
Perform N Class Reads   /* N=2 for double-buffered Class I/O. */
        /* This ensures that several input buffers are available. As
         * the data arrives, it will fill one buffer and then the
         * next. If all goes well, more Class Read requests will
         * replenish the supply of buffers before the last one is
         * filled. */
Repeat
    Perform a Class Get, with wait
        /* To get a buffer full of data. Other buffers are still
         * available to the system, so no data should be lost. */
    Perform a Class Read
```

```
        /* To replace the buffer that was just retrieved by the Class
         * Get. This ensures that the system does not run out of
         * buffer space for incoming data. */
      Process the data that has just been retrieved
    Loop until you decide to stop
    Flush pending I/O requests and de-allocate the Class Number
```

In practice, the system tends to lose data even with multiple-buffered Class I/O. The problem is compounded when a program wants to intermix input and output operations, because the pending input requests must be cancelled before the output can occur. The following algorithm illustrates the process:

```
    Cancel all outstanding Class Read requests
    Perform a Class Write to send the output data
    Perform sufficient Class Reads to reinstate the buffers that
        have just been cancelled
    Perform sufficient Class Get operations to retrieve the buffers
        from the cancelled Read requests.
    Perform a Class Get to retrieve the buffer from the Write request.
```

If the output performed above solicits an immediate response from the external device, the input could arrive before the Class Read operations are performed, which would lead to some of the input data being lost. There seems to be no solution to this problem.

If it were possible to cancel just the currently executing Read request, then the solution would be simple:

```
    Perform a Class Write
        /* The output will not occur yet because there are still
         * Read requests ahead of it on the device queue */
    Repeat N Times          /* where N = 2 for double-buffering */
      Perform a Class Read
        /* Allocate a buffer which will be available as soon as the
         * output operation completes */
      Perform a Class Get with no wait
      If no data was available then
        Cancel the current Read operation
        Perform a Class Get with wait
      Else
        Ignore the data that was retrieved
      End If
```

```
        /* The above few lines ensure that the current read request is
         * completed, either normally or by force */
    End Do
        /* Now all the Read requests that were ahead of the Write have
         * terminated, so the Write operation can proceed. Immediately
         * after the write, the correct number of Read buffers will be
         * available. */
    Perform a Class Get with wait
        /* To retrieve the buffer used in the Write operation. */
```

The above algorithm would ensure that the system had a read request pending as soon as the write was complete, but data loss would still be possible due to the limitations of most I/O interfaces.

Mixing I/O control operations with multiple-buffered Class I/O has problems identical to those associated with output operations.

All the terminal ports on our HP-A900 computer are controlled by HP-12040B eight-channel multiplexer (or MUX) cards [HP-12040], and use the interface driver **IDM00** and the device driver **DD*00**. This equipment has an input type-ahead buffering facility, which is essential to the Connect program because there is no other viable method of avoiding input data loss.

The Connect program assumes that the communications line it is using is connected to one of the eight ports on an HP-12040B MUX card, using the drivers mentioned above. The low level routines that actually interact with the drivers are separated from the rest of the software, so it would be relatively simple to adapt the code for other device interfaces, provided they allow input buffering and XON/XOFF flow control. Such adaptation has not been necessary because the Department's HP-A900 computer uses the HP-12040B MUX cards exclusively.

Following the recommendations in [RTE-DRM], the subroutine that handles the lowest level input attempts to read 254 characters at a time, and terminates the read request prematurely if necessary. Input is retrieved one character at a time from a buffer maintained by the software, until an entire line has been read or a timeout occurs.

Input buffer overflows can still occur, especially if the external device sends a large burst of data. For example, when attempting to create a connection through the CASE network, the network sends a three-line message that has been known to cause input buffer overflows.

The operating system reports a buffer overflow by printing a *Transmission Error* message on the user's terminal and cancelling the current read operation. This action leads to more data than necessary being lost, but there appears to be no way of making the system treat the buffer overflow in any other way.

The additional speed attained by the Connect program when using a binary command file instead of a printable command file greatly reduces the likelihood of input buffer overflows. In practice, buffer overflows have never been noticed when binary command files have been used.

Timeouts and WAIT commands have a resolution of 10 milliseconds, which is the resolution of the system time-base interrupt generator.

A FLUSH command that includes the optional delay period will not work exactly as specified in section A7.5. It will discard the current contents of the type-ahead buffer, perform a read operation with a timeout equal to the specified delay period, and ignore the data returned by the read operation. This can lead to incoming data being ignored for a period shorter than that specified in the FLUSH command.

Before opening the device, the program verifies that the specified LU number is valid, and that no user session is attached to the device.

The Connect program does not lock the device (using the system LURQ subroutine). The operating system's LU locks restrict the device availability to the program that created the lock, but other programs may bypass the lock if they know a 16-bit key. If the Connect program did lock the device in this way, other programs that did not know about the key would not be able to use the device. For this reason, locking the device appears to be undesirable.

### A7.9.8. **HP-A900 Terminal I/O**

All I/O to the user's terminal is done using Fortran READ and WRITE statements. To suppress end-of-line characters in a WRITE operation, the usual RTE-A method is used. This involves using an underscore character as the last character in an output operation. For example,

```
    write (stdout,'(A,1H_)') 'Text '
```

will output the five characters "Text ", without beginning a new line.

### A7.9.9. **HP-A900 Lock Files**

The MakeLockFile and KillLockFile subroutines on the HP-A900 are written in Fortran, using RTE-A FMP calls (which are the standard file management calls).

Files must be deleted by name, but it is not possible for one program to delete a file that another program is using. If a program attempts to delete a file that another program has open, the attempt fails. The MakeLockfile subroutine checks for this eventuality, as described in section A7.6.6, and will function correctly.

A7.9.10. **HP-A900 Installation and Command Line Arguments**

The executable code for the Connect program resides in the standard directory for executable programs. The file is named **/PROGRAMS/CONNECT.RUN**. This file is automatically executed when a user types the command **CONNECT**. The access permissions for the program are set so that any user can execute the program.

When a program is started under RTE-A, it can be passed a set of five integer valued arguments and a character string of up to 256 characters. The integer arguments can be retrieved by the system **RMPAR** subroutine, and the character arguments can be retrieved by an **EXEC 14** call. There are also various other ways of retrieving the character arguments, such as the system **GETST** subroutine or the Fortran library functions **RCPAR** and **RHPAR**.

The standard command line interpreter, **CI**, automatically sets the integer parameters to suitable values based on the command line typed by the user. It does this by breaking the command line into arguments, which are separated by commas. The first five of the arguments correspond to the five integer parameters. If a character argument consists entirely of digits, the number that it represents is used for the corresponding integer argument. If a character argument is absent, zero is used for the associated numeric argument. Otherwise, the ASCII code for the first two characters in the argument is stored into the integer argument.

CI usually converts the command line to upper case and add commas between the arguments, although this action can be suppressed. The name of the program is also included in the character string argument passed to the program. For example, if a user types following command:

```
connect commandfile dev=57
```

then the following character string argument will be passed to the Connect program:

```
RU,CONNECT,COMMANDFILE,DEV=57
```

It is possible to prevent the insertion of commas and the conversion to upper-case. The simplest way of doing this is to enclose the arguments in reverse quotation marks, as follows:

```
connect `commandfile dev=57`
```

In this case, CI would pass the following character string to the program:

```
RU,CONNECT,commandfile dev=57
```

If one program schedules another program directly, using a suitable EXEC call, there will not necessarily be any correspondence between the integer parameters and the character string parameter. The character string parameter also need not include the RU command and the program name, but it is conventional to include at least the first two commas. The invoked program will then know that its command line arguments begin after the second comma in the character string.

The Connect program needs to use the character string argument, and is not concerned with the integer arguments. The **GetRunstring** subroutine uses EXEC 14 to get the arguments into an integer array, which is **EQUIVALENCE'd** to a character variable. The subroutine then returns the part of the string after the second comma.

The Connect program treats commas and blanks in its command line as equivalent (except within quoted character strings). It also ignores the difference between upper-case and lower-case keywords and file names (but values assigned to symbols in the command line are case sensitive). It is safest to enclose the entire command line passed to the program in reverse quotation marks, to protect it from modification by the command interpreter, CI, but this is essential only if lower-case characters or blank spaces are required.

### A7.9.11. **HP-A900 Exit Status**

Just before terminating, the program calls the system **PRTN** subroutine, with a first argument value of 0 (for success), 1 (for too many steps) or 2 (for ABORT or other failure). It then does an **EXEC 6** call to terminate.

### A7.9.12. **HP-A900 File Names**

The *Device Information File*, or *INFO* file, is named **/CONN/DEVICE/INFO**. The log file is named **/CONN/DEVICE/LOGFILE**. The standard directory for command files is **/CONN/SCRIPTS**.

### A7.9.13. **VAX-750 Device I/O**

On the VAX-750, low level routines that handle I/O to the connection device are written in VAX MACRO assembly language. VMS system calls (especially SYS$QIOW) are used to perform all the device I/O.

VAX Record Management Service (RMS) calls may be used instead of system I/O calls, and RMS provides some device independence; the same RMS calls can often be used to access both disk files or terminals. A version of the device I/O subroutines used by the Connect program was written to use RMS instead of system I/O calls, and the results were satisfactory. RMS provides no way of changing a terminal's configuration, however, so system calls were needed for that task. It was therefore decided to use system calls throughout.

When the program tries to open the device, it checks that the named device is a terminal, that the user has permission to perform both input and output, that the device is not an operator's terminal, and that it is not spooled. If these tests pass, the device may be used.

The following device modes are set: XON/XOFF handshake in both directions, no echo, raw input, type-ahead. The original device mode settings are restored when the program terminates.

Input with timeout is accomplished by using the **IO$M_TIMED** option in the **IO$_READVBLK** read operation (which is one of the operations available in the **SYS$QIO** system call). A version of the subroutines was written that did not use the IO$M_TIMED option, but instead set a timer trap just before performing a read request and cancelled the read request if the timer expired before the read operation finished. The alternative version worked adequately, but the simpler method of setting the IO$M_TIMED option is the one currently in use.

Input is read one character at a time, until the software determines that an entire line has been read.

Input timeout periods are specified to a resolution of 1 second. Delay periods for the WAIT command are specified to a resolution of 100 nanoseconds.

A FLUSH command that includes the optional delay period will not work exactly as specified in section A7.5. It will discard the current contents of the type-ahead buffer, perform a read operation with a timeout equal to the specified delay period, and ignore the data returned by the read operation. The period during which input is ignored may be less than the period specified in the FLUSH command.

The device is not locked (using the system **SYS$ALLOC** service) by the Connect program. DCL command files that invoke the program may find it useful to use the DCL **ALLOCATE** command to ensure exclusive use of the device. The device protection must be set in such a way that users who are permitted to use the Connect program can allocate the necessary devices.

### A7.9.14. **VAX-750 Terminal I/O**

All I/O to the user's terminal is done using Fortran READ and WRITE statements. The non-standard "$" format descriptor is sometimes used to suppress the end-of-line characters produced by a WRITE statement. For example,

```
    write (stdout,'(A,$)') 'Text '
```

will output the five characters "`Text `", without beginning a new line. Unfortunately, this does not work in the same way as it does on some other systems. A Fortran WRITE operation to a terminal on the VAX-750 outputs a carriage-return (CR) and line-feed (LF) character at the start of the write operation, then it outputs the data, and finally it outputs a CR. The "$" format descriptor suppresses the CR at the end of the write operation, but does not suppress the CR and LF at the start of the next write operation. This means that several successive WRITE statements using this option will not all write to the same line on the display screen, as would be the case on some other systems.

There is a non-standard feature of the VAX Fortran OPEN statement that allows one to specify `CARRIAGECONTROL='NONE'` (or `'FORTRAN'`, which is the default used by VAX Fortran, or `'LIST'`, which corresponds to the default used by many other languages). The file **SYS$OUTPUT**, which is the standard output file for a program on the VAX, and is usually connected to the user's terminal, could be opened with the `CARRIAGECONTROL='NONE'` option. Fortran WRITE statements would then output only the specified data, without any CR or LF characters. The program could explicitly add CR and LF characters to start a new line when necessary. The potential gains from making such modifications to the program are small, because the only benefit would be a slightly better looking screen display, so the modifications have not been made.

### A7.9.15. **VAX-750 Lock Files**

The MakeLockFile and KillLockFile subroutines on the VAX-750 are written in VAX MACRO assembly language, using RMS calls (which are the standard file management calls).

A program must have a file open in order to delete it, and the deletion is performed as the program closes the file. If a program attempts to delete a file that another program has open, the first program closes the file successfully, but does not delete it. This action is acceptable, as discussed in section A7.6.6.

### A7.9.16. **VAX-750 Installation and Command Line Arguments**

A program under the VMS operating system may be run in its own separate *process* or *sub-process*, or may be run in the same process (or sub-process) as a command interpreter. If a program is run in its own process, there is no way for the program to be passed any command line arguments. If the program is run under the control of a command interpreter, the command interpreter may provide a way of passing command line arguments to the program.

The standard command interpreter is called **DCL** (Digital Command Language). There are three ways in which a user may instruct DCL to run a program. The **RUN** command is the simplest of these methods, and does not allow for command line arguments to be passed to the program. The second method, running the program as a *Foreign* program, allows the program to retrieve any command line arguments using the **LIB$GET_FOREIGN** system library function [VAX-RTL]. The most complicated method involves creating a text file that contains a *Command Definition* in a special format. The command definition must then be converted to a binary format and integrated into the command interpreter's internal tables, using the **SET COMMAND** command. When the command is used, the program can retrieve its arguments using the **CLI$...** subroutines [VAX-CDU]. This last method also requires the syntax of the command line arguments to match the syntax usually used for DCL commands.

The Connect program uses the second method described above (running the program as a *Foreign* program). The usual way of running a Foreign program from DCL involves defining a *Symbol* with a suitable translation. There is no DCL command that will run a Foreign program directly. On the Department of Electronic Engineering's VAX-750, the executable file CONNECT.EXE resides in a directory whose logical name is SYS$UTIL. The symbol **KONN*ECT** is defined as

```
KONN*ECT :== $ SYS$UTIL:CONNECT
```

The star (*) in the symbol name indicates that the full name (KONNECT) may be abbreviated to KONN (or to KONNE or KONNEC). The name begins with the letter **K** to distinguish it from the standard DCL **CONNECT** command. The first dollar sign ($) in the symbol definition identifies the program as a foreign program. When a user types a command whose first word is **KONNECT**, DCL invokes the Connect program in such a way that it can retrieve the remainder of the command line with a call to LIB$GET_FOREIGN. The access permissions for the executable file are set in such a way that any user may execute the program.

### A7.9.17. VAX-750 Exit Status

The program terminates by using the **SYS$EXIT** system call, with an argument value of **SS$_NORMAL + STS$M_INHIB_MSG** (for successful completion), **SS$_CONNECFAIL** (for too many steps), or **SS$_ABORT** (for ABORT or other failure). In addition to the message printed by the Connect program when it fails, DCL will print a message that corresponds to the program's exit status. This message is suppressed when the program is successful.

It is possible to define customised exit status codes, and corresponding error messages, but this is best done on a system-wide basis (instead of on a per-program basis) and has not been done for the Connect program. The exit codes that were chosen are those whose predefined messages correspond most closely to the error condition encountered. The message will be *Connect to network object timed out or failed* if the connection fails because too many steps were attempted in the command file, or will be *Abort* if the program aborts for some other reason.

### A7.9.18. VAX-750 File Names

The *Device Information File*, or *INFO* file, is named **CONN$DEVICE:INFO**. The log file is named **CONN$DEVICE:LOGFILE**. The standard directory for command files is **CONN$SCRIPTS:**. Suitable translations for the logical names **CONN$DEVICE** and **CONN$SCRIPTS** must be placed in the system logical name table when the system is booted up.

## Appendix A8. Example Connect Command Files

This appendix lists the TOCASE, CASEQ and FROMCASE command files used by the Connect program to make and break connections through the CASE network. Identical copies of these command files are used on all three computers that support the Connect program.

### A8.1. The TOCASE Command File

```
* @(#)tocase.con 1.3 87/09/24
* File: tocase.con
* Command file for connection through CASE to $DEST

* convert destination to uppercase
      set dest $dest FOLD
* check valid invocation
      if $dest = "" goto bad_invocation
* set max step count
      say '<using device ' $DEV ' to connect to ' $DEST '>\l'
      maxsteps 200
* start by sending DC1
      send ^q
start:  ;say <start>
* ignore any junk
ignore_junk:    ;say <ignore_junk>
      receive reply show timeout 0.2 goto send_ctl_T
      goto ignore_junk
send_ctl_T:     ;say <send_ctl_T>
* send ^t, return
      say '<sending control-T>'
      send ^t
      wait 0.5
      send \r
check_select:  ;say <check_select>
* look for SELECT A SERVICE message
      receive reply show timeout 3 goto start
      set reply $reply fold
      if $reply contains SELECT goto check_sel_junk
      goto check_select
check_sel_junk:  ;say <check_sel_junk>
```

```
* make sure nothing else follows SELECT prompt
      receive reply show timeout 0.5 goto send_dest
      goto check_select
send_dest:  ;say <send_dest>
* send destination, return
      say $dest \1
      send $DEST
      wait 0.5
      send \r
* special processing if $DEST = 'Q'
;;      if $DEST = 'Q' goto get_result_Q
get_result:  ;say <get_result>
* look for COM response
      receive reply show timeout 8 goto start
;      if $reply = "" goto start
check_result:
      if $reply contains 'COM' goto check_com_junk
* queue if OCC response
;;      if $reply contains 'OCC' goto set_dest_Q
      if $reply contains 'OCC' goto abort
* abort if error message
      if $reply contains 'ERR' goto abort
      if $reply contains 'DER' goto abort
      if $reply contains 'INV' goto abort
      if $reply contains 'NP' goto abort
      if $reply contains 'NA' goto abort
* check for another SELECT prompt
      set reply $reply fold
      if $reply contains SELECT goto check_sel_junk
* loop if message not understood ("MOM" will do this)
      goto get_result
check_com_junk:  ;say <check_com_junk>
* check for funny stuff after COM
      receive reply show timeout 0.5 goto success
      if $reply = '\n' goto check_com_junk
      if $reply = '' goto check_com_junk
      goto check_select ; there was some junk
success:  ;say <success>
* successful
```

```
      say <connected>\l
      finished
abort:  ;say <abort>\l
* abort
      abort

* bad invocation
bad_invocation:
      say "CONNECT TOCASE: must specify DEST"\l
      abort
```

A8.2. **The CASEQ Command File**

```
* @(#)caseq.con 1.2 87/08/24
* File: caseq.con
* Command file for connection through CASE to $DEST
*
* This command file will QUEUE the connection request if the CASE
* gives an OCC error for the requested destination.

* convert destination to uppercase
      set dest $dest FOLD
* check valid invocation
      if $dest = "" goto bad_invocation
* set max step count
     ;say '<using device ' $DEV ' to connect to ' $DEST '>\l'
      maxsteps 200
* start by sending DC1
      send ^q
start:  ;say <start>
* ignore any junk
ignore_junk:    ;say <ignore_junk>
      receive reply      timeout 0.2 goto send_ctl_T
      goto ignore_junk
send_ctl_T:     ;say <send_ctl_T>
* send ^t, return
     ;say '<sending control-T>'
      send ^t
      wait 0.5
     send \r
check_select:  ;say <check_select>
* look for SELECT A SERVICE message
     receive reply      timeout 3 goto start
     set reply $reply fold
     if $reply contains SELECT goto check_sel_junk
     goto check_select
check_sel_junk:  ;say <check_sel_junk>
* make sure nothing else follows SELECT prompt
     receive reply      timeout 0.5 goto send_dest
     goto check_select
send_dest:  ;say <send_dest>
* send destination, return
    ;say $dest \l
     send $DEST
     wait 0.5
     send \r
* special processing if $DEST = 'Q'
     if $DEST = 'Q' goto get_result_Q
get_result:  ;say <get_result>
* look for COM response
     receive reply      timeout 8 goto start
;     if $reply = "" goto start
check_result:
     if $reply contains 'COM' goto check_com_junk
* queue if OCC response
     if $reply contains 'OCC' goto set_dest_Q
* abort if error message
     if $reply contains 'ERR' goto abort
     if $reply contains 'DER' goto abort
     if $reply contains 'INV' goto abort
     if $reply contains 'NP' goto abort
     if $reply contains 'NA' goto abort
* check for another SELECT prompt
     set reply $reply fold
     if $reply contains SELECT goto check_sel_junk
* loop if message not understood ("MOM" will do this)
     goto get_result
check_com_junk:  ;say <check_com_junk>
```

```
* check for funny stuff after COM
      receive reply      timeout 0.5 goto success
      if $reply = '\n' goto check_com_junk
      if $reply = '' goto check_com_junk
      goto check_select ; there was some junk
success:  ;say <success>
* successful
      ;say <connected>\l
      finished
abort:  ;say <abort>\l
* abort
      abort

* special things for QUEUEing

set_dest_Q: ;say <set_dest_Q>
* OCC received
      set old_dest $DEST
      set DEST 'Q'
* try to queue
      goto check_select
get_result_Q: ;say <get_result_Q>
* here just after sending 'Q'
* get QUEUED message
      receive reply      timeout 5 goto check_result_Q
      goto get_result_Q
check_result_Q: ;say <check_result_Q>
      if $reply contains 'QUEUED' goto get_res_after_Q
      goto abort
get_res_after_Q: ;say <get_res_after_Q>
* wait for another message
      receive reply      timeout 60 goto timeout_after_Q
* continue normal processing if no timeout
      goto check_result
timeout_after_Q: ;say <timeout_after_Q>
* been in queue for 60 seconds, try again
      goto get_res_after_Q
```

```
* bad invocation
bad_invocation:
      say "CONNECT CASEQ: must specify DEST"\l
      abort
```

## A8.3. The FROMCASE Command File

```
* @(#)fromcase.con 1.2 87/08/24
* File: fromcase.con
* Command file for disconnection from CASE port

* Set default delay = 1 second
  if $delay <> "" goto got_delay
  set delay 1
got_delay:

* delay, then send control-T
  wait $delay
  send ^t
  finished
```

## Appendix A9. Example Use of the KERNET Command

The following example illustrates a typical use of the KERNET command. In this example, a user on the HP-A900 wishes to transfer a file from the VAX-750 to the HP-A900. Some details have been omitted, but all the commands typed by the user are shown. Information typed by the user appears in `typewriter` style, information printed by the local computer appears in **boldface**, and information printed by the CASE network or the remote computer (and relayed by the Connect program or the Kermit program on the local computer) appears in *italics*. Explanatory information appears in ordinary type.

```
CI> kernet 57 750
                /* The KERNET command is given at the system
                /* CI prompt. The following lines are printed
                /* as the connection is made */
<using device 57 to connect to 750>
<sending control-T> \r\n
University of Natal Network. Electronic Engineering Node. \r\n
08.008\r\n
Please Select a Service.  Use HELP for more information. \r\n
750
\r\n
COM\r\n
<connected>
                /* Now the Kermit program on the local
                /* computer is automatically started */
HP-1000 RTE-KERMIT Version 1.97  <871110.2220>

 KERMIT-RTE_A requires EOL=13!

KERMIT-RTE is in remote-host mode; file transfers are ok
KERMIT-RTE is in local-host mode to LU 57 @ 9600 baud;  Parity = EVEN
Kermit-RTE> c
                /* The user gives the "C" command to enter
                /* Kermit's transparent mode */
[connecting to LU 57; return via "control-]" then "C"]
                /* The user's terminal is now effectively
                /* connected to the remote computer.
                /* Press RETURN to get a log-in prompt from the
                /* remote computer, then log-in. */
```

```
Username: BARRETT
Password:
                /* After some messages from the system (not
                /* shown here), the user is logged in to the
                /* remote computer. */
$ dir
                /* Get a directory of files on the remote
                /* computer */
Directory DUA0:[BARRETT]

  FORT.DIR;1       GRAPH.TXT;1      GRAPH.DIS;1      LIB.DIR;1
  LOGIN.COM;3      LOGIN.COM;2      LOGIN.COM;1      LOGIN_1.COM;1
  MACRO.DIR;1      MAIL.MAI;1       TEXT.DIR;1       UTIL.DIR;1

Total of 12 files.
$ kermit
                /* Run KERMIT on the remote computer */
VMS Kermit-32 version 3.2.075
Default terminal for transfers is: _TXA0:
KERMIT-32/EE750> server
                /* Place the remote Kermit program in SERVER
                /* mode, to simplify file transfer procedure.
                /* In server mode, the remote Kermit program
                /* gets its instructions in the form of packets
                /* transmitted by the local Kermit. The local
                /* Kermit will transmit the packetised commands
                /* in response to keyboard commands from the
                /* user. */
Kermit Server running on VAX/VMS host.
Please type your escape sequence to return to your local machine.
Shut down the server with the Kermit BYE command on your local machine.
```

```
                /* At this point, the user types a control-]
                /* character, followed by a C. This terminates
                /* the local Kermit program's transparent mode.
[back at KERMIT-RTE]
                /* Now the local Kermit is ready for more
                /* commands, and the remote Kermit is waiting
                /* for the local Kermit to tell it what to do */
Kermit-RTE> get login.com
                /* The GET command is used to get a file from
                /* the remote computer. */
   0/000 Warning: File will be renamed to LOGIN001.COM:::4:24
  54/000 Receiving LOGIN001.COM:::4:24
File transfer completed
                /* The file transfer was successful. The local
                /* Kermit renamed the incoming file because
                /* its name clashed with an already existing
                /* file. */
Kermit-RTE> finish
                /* The FINISH command causes the local Kermit to
                /* tell the remote Kermit that it must leave
                /* Server mode. */
Kermit-RTE> c
                /* Make the local Kermit go into transparent
                /* mode, so the user's terminal is effectively
                /* connected to the remote computer. */
[connecting to LU 57; return via "control-]" then "C"]
                /* The remote Kermit is still running, but is
                /* no longer in server mode */
KERMIT-32/EE750> ex
                /* Terminate the Kermit program on the remote
                /* computer */
$ eoj
                /* Logout from the remote computer. See
                /* comments below. */
 BARRETT     logged out at  8-JAN-1988 11:00:24.13
                /* Type control-] followed by C to terminate
                /* the transparent mode on the local Kermit. */
[back at KERMIT-RTE]
```

```
Kermit-RTE> ex
                /* Terminate the local Kermit program. The
                /* KERNET procedure automatically disconnects
                /* the network link. */
Disconnecting ...
                /* The local computer is now back at its
                /* system prompt. */
CI>
```

In the above procedure, special care must be taken when logging out from the remote computer. Most of the computers on EENET have been configured to transmit a *control-T* character when a user logs out. This has the effect of closing the CASE network connection between the user's terminal and the computer, and is usually desirable.

When using KERNET, however, the user's terminal is only indirectly connected to the remote computer. A *control-T* character sent by the remote computer would break the link between the two computers (which is desirable). It would also be relayed, by the local computer's Kermit program (which is in transparent mode), to the link between the local computer and the user's terminal. This would have the effect of breaking the link between the user's terminal and the local computer, which is undesirable.

To avoid this problem, it is necessary either to prevent the remote computer from sending a *control-T* character, or to ensure that the local Kermit program is no longer in transparent mode when the remote computer does send a *control-T* character.

In the above example, the **EOJ** command was used instead of the normal **LOGOUT** command. This prevents the VAX-750 from sending a *control-T* character. Another possible solution would be to define a command that delays for several seconds before logging out, giving the user time to terminate the local Kermit's transparent mode.

## Appendix A10. Software on the Gerber IDS-80

This appendix describes the network access software written for the Gerber IDS-80. This includes the CASE program, various subroutines that simplify the task of other programs that wish to schedule the CASE program, and the modifications made to the SPMON print spooler and the PLTHP plotter driver programs.

### A10.1. The CASE program

The CASE program provides the basic capability for the Gerber IDS-80 computer to create and close connections through the CASE network.

To create a connection through the CASE network, the following command can be used:

`RUN,CASE,` *syslu* **,** *destination* **,** *debuglevel*

where *syslu* is the system logical unit number (*LU*) of the serial port that must be used for the connection, *destination* is the name of the network destination to which the port must be connected, and *debuglevel* is an optional integer parameter used to control the display of information intended for debugging. If the *debuglevel* parameter is not given, or if it is specified as zero, no debugging information will be displayed. Values 1 and 2 specify different degrees of debugging information.

To close a connection, the following command is used:

`RUN,CASE,` *syslu*

In both cases, the program terminates with an exit status of zero for success, or 1 for failure. The system subroutine PRTN is used to return the program's exit status. If the program fails, an error message is printed on the user's terminal.

The CASE program is loaded as a system permanent program. This means that it can be executed at any time, and that multiple copies, or *clones*, of the program can execute simultaneously. The DMS RUN command used in the examples above automatically clones the program.

The program assumes that the device used for the network connection is one of the four ports on a version 2 RMUX card [Pepl88]. The RMUX card and the associated driver software allows input type-ahead buffering and XON/XOFF flow control, which the CASE program relies on to avoid data loss.

The program makes no attempt to lock the LU that it uses for the network connection. Other programs that schedule the CASE program generally use the system LURQS subroutine to lock the LU. This method of locking does not prevent the CASE program from being able to access the LU.

## A10.2. The CSSKD and CSSKW Subroutines

The CSSKD and CSSKW subroutines can be used by other software to schedule the CASE program. CSSKD schedules the CASE program, waits for it to terminate, and returns a flag to indicate success or failure. CSSKW is similar, except that it repeatedly re-tries until the CASE program reports success.

These subroutines are declared as follows:

```
SUBROUTINE CSSKD (SYSLU,DEST,DLEN,STATUS)
INTEGER SYSLU,DEST(8),DLEN,STATUS

SUBROUTINE CSSKW (SYSLU,DEST,DLEN,DELAY)
INTEGER SYSLU,DEST(8),DLEN,DELAY
```

*SYSLU* is the system LU number to be used for the connection. *DEST* is an integer array containing the CASE destination address (up to 16 characters). *DLEN* gives the actual length of the destination name.

The CSSKD subroutine builds a suitable command line argument string and schedules the CASE program. It then returns a non-zero status code if the CASE program could not be scheduled or if it did not complete successfully.

The CSSKW subroutine does not return a status code, it simply loops calling CSSKD repeatedly, until it is successful. The *DELAY* argument passed to CSSKW specifies how long to wait between connection attempts, in ten-millisecond units.

## A10.3. Device Descriptions and the CSCHK Subroutine

On the IDS-80 system, each system LU has a device type number and various type-dependent parameters. There is also a short text description associated with each system LU [IDS-SRM1]. There is no common convention regarding the information that is placed in the text description. Gerber-supplied software that wishes to use a printer or plotter, for example, often checks the device type parameters to verify that the relevant device is of the required type.

It was decided to use a special convention in the text field of the device information database to identify the CASE network destination (if any) associated with a particular LU. This convention states that the text description will begin with the two characters "C=" if the device should be connected through the CASE network. The network destination is listed just after the "C=" characters. A semicolon (;) terminates the network destination name. Arbitrary other information may appear after the semicolon. The use of this convention provides a simple means for software to determine whether a particular system LU should be connected through the CASE network, provided that the LU will always be connected to the same network destination.

A subroutine called CSCHK was written to check for the special “C=dest;” information, and to return the network destination if the special information is found. This subroutine is declared as follows:

```
SUBROUTINE CSCHK (SYSU,DEST,DLEN)
INTEGER SYSLU, DEST(8), DLEN
```

*SYSLU* is the system LU number to be checked. *DEST* is an integer array into which CSCHK will store the network address that should be used for the device, if any. *DLEN* returns the length of the network address, or zero if the device does not need to be connected through the CASE network.

## Appendix A11. The "case" Printer Interface on the HP-9000

The standard print spooling system on the HP-9000 (**lpr**) allows user-defined interface programs to be installed, to handle access to specific printers. An interface program named **case** was developed to allow assess to printers connected through the CASE network. This interface program is in the form of a *shell script* (a command file interpreted by the system's standard command interpreter, the *shell*), and is based on a standard interface program designed for a dumb printer.

### A11.1. Introduction

Print jobs submitted by users are added to one or more print queues. The **lpr** or **lp** program is used to submit a print job. Several options can be passed to **lpr**, including a title for the print job, the number of copies, the printer or class of printers to be used, and various printer- or class dependent options.

A program named **lpsched** manages the print queues. When a printer is available and a suitable print job is waiting, **lpsched** invokes the appropriate interface program, passing it various command line arguments describing the print job. These arguments are order-dependent, and have the following meanings (**$0** is the name of the interface program, **$1** refers to the first argument, **$2** to the second, etc.):

| | |
|---|---|
| $1 | Print request ID number |
| $2 | User name |
| $3 | Title supplied by **-t** option of **lpr** |
| $4 | Number of copies required |
| $5 | Printer- or class dependent options specified by **-o** option of **lpr** |
| $6 (and others) | Names of files to be printed |

When the interface program is scheduled, its standard output is connected to the device that is used to access the relevant printer. The **case** interface program needs to know the name of this device, because the name needs to be passed as an argument to the Connect program. The device name is stored (by **lpadmin**) in the first line of the file **/usr/spool/lp/member/***name*, where *name* represents the name of the printer. The interface program simply reads this file to get the device name.

When a printer is connected through the CASE network, the interface program needs to know the network address of the printer, as well as the name of the device (on the local computer) that will be used for the connection. The default network destination address for the printer is hard coded into the interface program (where it is easily modified, because the interface program is a shell script). The network destination address may be overridden when a print job is submitted, as described below.

## A11.2. Printer Dependent Options

When **lpr** is used to submit a print job, its **-o** option may be used to specify some printer-dependent options that will eventually be passed to the interface program. The options and the -o flag that introduces them must be passed to lpr as a single *word*, in the sense used by the Unix shell [HP-UX, section sh(1)], which means that quotation marks must be used if the set of options includes blank characters. For example, a user might give the following command to the standard command interpreter (the Bourne shell):

```
lpr -o"d=8.34 noinit" file
```

The character string "**d=8.34 noinit**" (without the quotation marks) would then eventually be passed as the fifth argument to the interface program. The interface program treats blanks as delimiters between options, so it would treat the character strings "**d=8.34**" and "**noinit**" as individual options.

The following printer-dependent options are understood by the interface program for the **case** printer:

| | |
|---|---|
| **d=***dest* | Use a network destination address other than the default. |
| **nohead** | Do not print a header at the start of the print job. |
| **nofeed** | Do not perform a form-feed at the end of the print job (or between files, if there are several files in the print job). |
| **noinit** | Do not send any control sequence to initialise the printer. |
| **init=***string* | Send the specified initialisation sequence to the printer at the start of the print job. The initialisation sequence must be given in a form acceptable to the standard **/bin/echo** program. For example, a print request that specified **-o'init=\033X'** would result in an ASCII *ESC* character (octal code 033) followed by an **X** character being sent to the printer. In this example, the quotation marks are required to prevent the shell from interpreting the backslash character. |

There are also several other options that allow predefined initialisation sequences to be sent to the plotter. These options consist of a printer name, a dash, and a number. The printer name is necessary because the type of printer that is physically connected to the network is not known by the interface program. The number after the printer name specifies the number of characters per inch. The following options of this type are currently available. The default, if no initialisation is explicitly specified, is honeywell-10.

| | |
|---|---|
| **anadex-10fast** | Anadex DP-9620A, 10 cpi, fast mode. |
| **anadex-10** | Anadex, 10 cpi, slow (better quality) mode. |
| **anadex-12** | Anadex, 12 cpi. |
| **anadex-15** | Anadex, 15 cpi. |
| **anadex-16.4** | Anadex, 16.4 cpi. |
| **honeywell-10** | Honeywell R32, 10 cpi. |
| **honeywell-16.66** | Honeywell, 16.66 cpi. |

## A11.3. Connecting Through the CASE Network

The **case** interface program uses the Connect program with the *caseq* command file to make a connection through the CASE network to the required printer. It loops until the connection is made successfully. After an unsuccessful connection attempt, the interface program delays for a short period before retrying. The length of the delay starts at ten seconds, and progressively increases as more failures occur, to a maximum of two minutes.

Once the connection has been made, the interface program may send a special character sequence to set the printer to a particular mode, according to the options specified by the user. It then outputs a standard header and sends the files comprising the print job. Finally, a form feed is sent.

Now the network connection must be broken. Some of the data destined for the printer may not yet have reached the printer (because it may still reside in buffers in the network). If the connection is broken immediately, this data will be lost. To guard against this problem, the interface program delays for 60 seconds before disconnecting. No delay can guarantee that there will be no data loss, but in practise the 60 second delay is usually sufficient.

## A11.4. Cancelling a Print Job

When a user uses the **cancel** command to abort a print job that has already begun, a signal is sent to the interface program responsible for that print job. The signal used is known as SIGTERM, or signal number 15, which is the signal usually used to allow a process to terminate gracefully. If the interface program takes no special action to trap this signal, the signal will cause the program to abort. This is the desired response in most cases, but in the case of the **case** interface program it could leave a connection through the CASE network unbroken. The **case** interface program traps the signal, and ensures that the CASE network connection is broken correctly.

## Appendix A12. The LPR Print Spooler on the HP-A900

This appendix describes the method used by the **LPR** print utility on the HP-A900 computer to access printers connected through the CASE network. **LPR** was originally written by D. Abbot for the HP-1000F computer, in 1986, and has been modified both to operate on the HP-A900 computer and to allow access to printers through the CASE network.

The LPR program itself does not do any printing. It simply verifies the command line arguments passed to it, schedules the PRNT program (which will actually do the printing) and passes a block of data to the PRNT program (using program-to-program CLASS I/O). The LPR program is now free to return control to the user, although the print job has not yet been completed.

The PRNT program is told the name of the printer to be used. The control codes and other parameters required to drive a particular printer are listed in a data file that resides in a directory named **/PRINTER**. Each printer has its own data file in this directory. The file name is the same as the printer name, but with the extension **.PR** added. For example, the data file **/PRINTER/DEFAULT.PR** contains the parameters that are required for the printer named **DEFAULT** (which is the printer that is used if the user does not explicitly request another printer).

The LU number that must be used for communication with the printer is not listed in the "**.PR**" data file. Instead, the PRNT program runs a program named GETPRLU which determines the LU number and returns it to the PRNT program. The original implementation of GETPRLU simply looked up the printer name and LU number in a data file called **/PRINTER/PRLU.DAT**. GETPRLU returns the LU number as the first **PRTN** parameter, and the PRNT program uses the system **RMPAR** subroutine to retrieve its value.

The format of the data file used by the GETPRLU program was augmented to include a network destination address as well as the printer name and LU number that were originally present. The CASE destination appears simply as a third item in records that apply to printers connected through the CASE network. The following is an extract from the **/PRINTER/PRLU.DAT** file:

```
DEFAULT 6
HP2632  6
ANADEX  57 ANADEX
```

Here the default printer is connected to LU 6, HP2632 is another name for the default printer, and the printer named ANADEX is connected via LU 57 to the CASE network destination named ANADEX.

The modified PRNT program schedules GETPRLU twice, initially to connect to the printer and finally to disconnect. PRNT passes two command-line arguments to GETPRLU. The first argument is the printer name and the second argument is either 1 (when a connection must be made) or 0 (when a connection must be broken).

The GETPRLU program was completely re-written to use this new data file format. If the **/PRINTER/PRLU.DAT** data file indicates that the printer must be connected through the CASE network, then GETPRLU schedules the Connect program to create or break the connection, as appropriate. When attempting to make a connection through the network, GETPRLU will loop until the Connect program is successful. This action is completely transparent to the PRNT program, and to the user.

## Appendix A13. Using the CASE Network From a PC

### A13.1. Access to Shared Peripherals

Peripherals that are connected to the CASE network may be used by a PC that is also connected to the network. Standard *terminal emulator* or *communications* programs may be used to create and close the required connections through the network.

For example, to use a plotter connected through the EENET, a PC user would need both a program that can drive the particular brand of plotter and a communications program that would allow the initial connection to be set up. First, the communications program would be used to generate a *connect event*. The CASE network would respond with its usual prompt, and the user would type the destination address of the desired plotter. If the connection is successful, the communications program can be terminated and the plotting program can be executed. Finally, the communications program would be used to generate a *disconnect event*.

One problem with this approach is that most plotting programs on the IBM PC and compatible machines do not support the XON/XOFF handshake used by most ports on EENET.

Ports on the network can be configured to use the hardware handshake usually used by the PC, and to translate this to XON/XOFF handshake for use by other ports. Unfortunately, ports cannot be set up in this way individually; the flow control configuration applies to all the ports sharing an LSC card in the DCX-850 or a terminal concentrator like the DCX-815. A group of four or eight ports would therefore all have to be re-configured simultaneously in order to support hardware handshake.

### A13.2. File Transfers

A PC user may transfer files between the PC and computers connected to the CASE network using standard *terminal emulator* or *communications* programs.

The PC user must run the communications program, use it to generate a CASE network *connect event*, connect to the desired host computer, and log-in. At the end of the session, he must logout from the host computer, ensure that the CASE network connection has been closed, and exit from the communications program on the PC.

During the session on the host, files may be transferred. Many communications programs allows incoming data to be logged to disk; this means that to transfer a text file from the host to the PC, all that is necessary is to enable the program's capture mode, and to make the remote computer type the file so that it appears on the display screen. Many communications programs also allow text to be sent from the PC's disk to the host computer. Files may then be sent from the PC to the host by configuring the host so that incoming data (from what the host believes is the user's keyboard) is saved in a file.

The file transfer methods just described are very primitive. They work only for text files, and are prone to data corruption or loss. A much more reliable method of file transfer between a PC and another computer is to use the Kermit protocol [Kerm84] [Kerm85] which was designed specifically for this purpose. A PC user would typically use the PC version of the Kermit program to create a connection through the network to the host computer and to log-in. The Kermit program on the host would then be started. File transfers in either direction could then be performed. After transferring the desired files, the user would terminate the remote Kermit program, logout from the host, ensure that the network link is broken, and terminate the PC Kermit program.

## Appendix A14. Non-standard Fortran Features

The Connect program is written in an extended version of Fortran-77. (Standard Fortran-77 is defined by ANSI standard X3.9-1978.) Much use is made of the language extensions mentioned below, which are available on all three computers that the Connect program currently runs on. Many of the extensions are part of the USA Department of Defence standard MIL-STD-1753. All of the extensions described here are common on minicomputers. Some other non-standard features, specific to a particular computer, may be used in versions of the program designed for that computer; these are not mentioned here.

### A14.1. MIL-STD-1753 Extensions

**IMPLICIT NONE** statements, used to force the compiler to report undeclared variables as errors.

Block **DO**-loops, without statement label numbers, using the **ENDDO** statement to terminate the loop.

**DO WHILE** loops, without statement numbers, terminated by an **ENDDO** statement.

**INCLUDE** statements, to include the contents of a separate file within a program unit being compiled. This is used to include declarations for variables in COMMON blocks.

### A14.2. Other Extensions

Lower-case characters are used in Fortran keywords and in identifier names.

Identifier names longer than six characters are used.

Comments appear after an exclamation mark (!) at the end of statement lines or continuation lines.

Integer variables are sometimes declared as **INTEGER*2** or **INTEGER*4**, to specify how much storage space should be allocated to them. This feature is used in very few cases.

## References

[Barr86] A.P. Barrett, "GRAPH User Manual" (unpublished), Department of Electronic Engineering, University of Natal, 1986.

[CASE-815] *CASE DCX 815 User Guide*, issue 4, Computer and Systems Engineering plc., Watford.

[CASE-840] *CASE 840 Data Concentrating Exchange User Guide*, issue 2, Computer and Systems Engineering plc., Watford.

[CASE-850] *CASE DCX 850 User Guide*, issue 5, Computer and Systems Engineering plc., Watford.

[CASE-LSC] *CASE LSC 3, 3A and 4 Modules User Guide*, issue 2, Computer and Systems Engineering plc., Watford.

[EIA69] "EIA Standard RS-232-C: Interface Between Data Terminal Equipment and Data Communications Equipment Employing Serial Binary Data Exchange", Electronic Industries Association, 1969.

[Elov74] H.S. Elovitz and C.L. Heitmeyer, "What is a Computer Network?", in *Tutorial: Distributed Processing*, 3rd edition, B.H. Liebowitz and J.H. Carson, eds., IEEE Computer Society Press, Silver Spring, 1980.

[Hals85] F. Halsall, *Introduction to Data Communications and Computer Networks* (book), Addison-Wesley, 1985.

[HP-12040] "HP 12040B 8-Channel Asynchronous Multiplexer Subsystem Installation and Reference Manual", Hewlett-Packard Company, Roseville, 1983.

[HP-UX] *HP-UX Reference*, Hewlett-Packard Company, Fort Collins, 1983.

[IDS-SRM1] *IDS-80/SRM-1 User Reference Manual*, volume 1 "Operating Environment", Gerber Systems Technology, Inc.

[Kerm84] *Kermit Protocol Manual*, 5th edition, F. da Cruz, ed., Columbia University Centre for Computing Activities, New York, 1984.

[Kerm85] *Kermit User Guide*, 6th edition, F. da Cruz, ed., Columbia University Centre for Computing Activities, New York, 1985.

[Lam84] S.S. Lam, "Data Link Control Procedures", in *Tutorial: Principles of Communication and Networking Protocols*, IEEE Computer Society Press, Silver Spring, 1984.

[Loom83] M.E.S. Loomis, *Data Communications* (book), Prentice-Hall, Englewood Cliffs, 1983.

[Mvin87] A.M. Mvinjelwa, "File Transfer Between Computers" (B.Sc.Eng. dissertation), Department of Electronic Engineering, University of Natal, Durban, 1987.

[Pehr73] D.L. Pehrson, "Interfacing and Data Concentration", in *Computer-Communication Networks*, N. Abramson and F.F. Kuo, eds., Prentice-Hall, Englewood Cliffs, 1973.

[Pepl88] R. Peplow, "An Intelligent Multiterminal Interface" (M.Sc.Eng. thesis, to be submitted), Department of Electronic Engineering, University of Natal, Durban, 1988.

[RTE-DRM] *RTE-A Driver Reference Manual*, Hewlett-Packard Company, Cupertino, 1985.

[RTE-Ori] *Orientation to RTE-A/VC+ for A-Series Users*, Hewlett-Packard Company, Cupertino, 1983.

[RTE-PRM] *RTE-A Programmer's Reference Manual*, Hewlett-Packard Company, Cupertino, 1985.

[RTE-Uti] *RTE-A Utilities Manual*, Hewlett-Packard Company, Cupertino, 1983.

[Stal85] W. Stallings, *Data and Computer Communications* (book), Macmillan, New York, 1985.

[Tane81] A.S. Tanenbaum, *Computer Networks* (book), Prentice-Hall, Englewood Cliffs, 1981.

[VAX-CDU] *VAX/VMS Command Definition Utility Reference Manual*, Digital Equipment Corporation, Maynard, 1986.

[VAX-DCL] *VAX/VMS DCL Dictionary*, Digital Equipment Corporation, Maynard, 1986.

[VAX-RTL] *VAX/VMS Run-Time Library Routines Reference Manual*, Digital Equipment Corporation, Maynard, 1986.

[Zimm81] H. Zimmermann and L. Pouzin, "The Standard Network Architecture Developed by ISO", in *Data Communications and Computer Networks: Proceedings of the IFIP (TC6)/CSI Conference on Networks 80*, North Holland Publishing Co., Amsterdam, 1981.