COMMUNICATION BETWEEN MICROCOMPUTERS
FOR SHARING OF
METEOROLOGICAL DATA

by
Robert L. Krawitz

SUBMITTED TO THE DEPARTMENT OF
DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF SCIENCE

at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1987

Copyright (c) 1987 Robert L. Krawitz

The Author hereby grants to M.I.T. permission to reproduce and to
distribute publicly copies of this thesis document in whole or in
part.

Signature of Author

Department of Electrical Engineering and Computer Science
May 18, 1987

Certified by
Professor Randall Dole
Thesis Supervisor

Accepted by
Professor Leonard A. Gould
Chairman, Department Committee on Undergraduate Theses

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 15 1987
LIBRARIES
COMMUNICATION BETWEEN MICROCOMPUTERS FOR SHARING OF METEOROLOGICAL DATA

by

Robert L. Krawitz

Submitted to the Department of Electrical Engineering and Computer Science on May 18, 1987 in partial fulfillment of the requirements for the degree of Bachelor of Science.

Abstract

The Simple Weather Transport Protocol (SWTP), which transfers weather data between an 8086-based computer running Concurrent DOS and a MicroVAX II running VMS is designed and implemented. SWTP allows existing DOS files, real time data, and VMS commands to be transferred to the MicroVAX over a single communication channel. Data transfer and command execution may be assigned priority levels depending on user preference. The system was only partially tested due to the unavailability of certain hardware modifications.

Thesis Supervisor: Professor Randall Dole
Title: Professor of Meteorology
# Table of Contents

Abstract 2  
Table of Contents 3  
List of Figures 4  
List of Tables 5  

## 1. Introduction 6  

## 2. Resources Available For the Project 8  
  2.1 Hardware Facilities 8  
  2.2 Software 9  

## 3. Basic Data Types 12  

## 4. Design of Zephyr/SWTP 14  
  4.1 The Final Design of Zephyr/SWTP 14  
  4.2 An Enhanced Design for Zephyr/SWTP 19  

## 5. Implementation of Zephyr/SWTP 25  
  5.1 Implementation on the LDP 25  
  5.2 Implementation on the VAX 35  

## 6. Analysis of Zephyr/SWTP 36  

## 7. Summary 43  

### Appendix A. Non-Interrupt Driven Transport Layer 44  

### Appendix B. A Testing Strategy for Zephyr/SWTP 50  

### Appendix C. Low-Level Extensions to SWTP 54  
  C.1 Packet Format 54  
  C.2 Flow Control 57  
  C.3 Signalling 59  
  C.4 Duplex Communications 60
List of Figures

Figure C-1: Sample SWTP Packet Format
List of Tables

Table 6-I: Maximum File Transfers at Selected Communication Rates and Queueing Capacities (K bytes) 38
Table 6-II: Time Required for Queue to Empty at Selected Communication Rates and Queue Lengths (seconds) 39
Chapter 1

Introduction

The Center for Meteorology and Physical Oceanography (CMPO) subscribes to a real time weather data service known as the Domestic Data Service (DDS) for educational and operational use. The CMPO currently uses a specially designed system called Zephyr, running on an 8086-based microcomputer, an LDP 2W to receive and process this data. The CMPO would also like to process this data in real time with a meteorological data analysis package called GEMPAK which runs on a MicroVAX II\(^1\) workstation. They would also like to be able to transfer files from the LDP to the VAX, and issue commands on the LDP that will run on the VAX. However, since only one communications port is available on the VAX, these three tasks must share the communication channel. The goal of this thesis is therefore to design and implement a communications system in software that will permit the VAX to receive both types of data over the same channel.

The LDP has a considerable base of software to massage the DDS data, which could not be duplicated on the VAX without an excessive investment in human resources. It also has a 85 megabyte (Mb) hard disk, of which very little is taken up by the operating system and Zephyr, which allows Zephyr to store several days' worth of real time data. On the other hand, only the VAX has the graphic and CPU resources necessary to run GEMPAK. It also possesses a 95 Mb cartridge tape drive which is more suitable for long term archiving of the data than any means directly available to the LDP. Since the archival data sets are produced by

\(^{1}\)VAX, MicroVAX, VAXstation, and VMS are registered trademarks of Digital Equipment Corp.
the massaging software on the LDP, it is necessary to transfer these data to the VAX for archiving.

There is only one communications port on the VAX that can be dedicated to communication with an external data source, so it is impossible to simultaneously connect the VAX to both the DDS and the LDP. Therefore, the LDP must send both real time data and archived data files to the VAX along with any control information necessary to process the data on the VAX. The Simple Weather Transport Protocol, (SWTP), as described in the rest of this thesis was designed and implemented to solve this problem.

The remainder of the thesis is organized as follows. Chapter 2 discusses the hardware and software resources available for this project, along with their limitations. Chapter 3 discusses the data handled by SWTP and Zephyr\textsuperscript{2}. Chapter 4 describes the design of two systems: an optimal communications protocol and the currently implemented version of Zephyr/SWTP, and Chapter 5 describes the actual implementation. Chapter 6 analyzes the implemented Zephyr/SWTP, and Chapter 7 summarizes the project and presents some suggestions for improvements. Appendix A describes an alternate implementation of Zephyr/SWTP that can be used if certain anticipated hardware facilities are not available. Appendix B describes a testing strategy for Zephyr/SWTP, including a discussion of testing already performed, and Appendix C presents some ideas on packet switching with respect to SWTP.

---

\textsuperscript{2}The combination of Zephyr and SWTP will be known as Zephyr/SWTP in this thesis.
Chapter 2

Resources Available For the Project

2.1 Hardware Facilities

The hardware available for Zephyr/SWTP consists of two computers, an LDP-2W running Concurrent DOS (formerly Concurrent CP/M)$^3$, and a VAXstation II RC running the VMS operating system. The LDP is based on the Intel 8086 microprocessor, and the VAXstation is based on Digital Equipment’s MicroVAX processor. One serial port on the LDP and one serial port on the VAX are available for use by Zephyr/SWTP, permitting only one communications channel between the two machines. This serial port can be configured to run at speeds up to 19,200 bits per second (bps).

The VAXstation includes a high-resolution bitmapped display unit (a QVSS) along with supporting software, which enable it to be used by terminal emulators, graphic packages, etc. It includes one 70 Mb disk drive, of which approximately 40 Mb is consumed by the operating system and essential utilities in the operating environment. It also includes 90 Mb streaming tape drive, which is used for backup and archival purposes, and a dot matrix printer with graphic capabilities. Two serial ports connect to user terminals.

The LDP contains one 85 Mb hard disk drive of which less than 20 Mb is used by the operating system and essential programs. It also contains one eight inch floppy disk drive which is used for backups of the system. There are at present three serial ports, one console port, and one printer port on the machine.

$^3$Concurrent DOS and Concurrent CP/M are trademarks of Digital Research, Inc.
One of the serial ports connects to a user terminal, one is connected to the incoming DDS line, and the last port is connected to the VAX and used by SWTP. The machine contains 768K bytes of RAM.

The serial ports are independently configurable as to bit rate and other communications parameters. The port connected to DDS can be configured to interrupt the processor when ready to transmit a character, but at the time of this writing the necessary work has not been done. The port connected to the VAX should be capable of interrupting the processor on transmit ready and received character available conditions in order for SWTP to work fully; at this time, it is believed possible with simple hardware modifications. This capability is not absolutely critical to the operation of the system, but its presence substantially increases the system’s performance.

2.2 Software

The basis for Zephyr/SWTP is the original version of Zephyr, which is a locally developed package for receiving data from the DDS line and storing it in an organized fashion. The Zephyr implementation on the LDP consists of the Zephyr kernel, a set of utilities that can retrieve data from files generated by the Zephyr kernel, other utilities that massage the data files, and a user interface to these utilities. Unless otherwise specified, we will use Zephyr to refer to the Zephyr kernel alone, since the utility packages are completely unaffected by the SWTP enhancements.

Concurrent DOS is a relatively simple operating system that runs on microcomputers. It is capable of supporting multiple concurrent processes, handling multiple I/O devices, and interprocess communication. Concurrent DOS
supports system queues, which permit processes to communicate with each other via named entities. System queues send data as fixed length messages, the length and maximum number of which are defined when the queue is created. Once a queue is created, it stays around and can be used by any process that knows the name of the queue, either for reading or writing. A mechanism exists to lock a queue for either reading or writing to prevent other processes from using the queue at critical times. The total maximum size of all system queues is defined at system configuration time, and can be changed by reconfiguring the kernel. Due to the large available memory on the LDP, it will probably be configured to between 128K and 256K bytes.

Concurrent DOS supports multiple simultaneous processes. Processes in Concurrent DOS can be either user processes or Resident System Processes (RSP’s), which are compiled directly into the operating system and run as part of the operating system kernel. There is a bug in Concurrent DOS which apparently limits the number of concurrent user processes to four; attempting to exceed this limit frequently causes the system to crash. This problem introduced implementation constraints on SWTP. At present, the only development software on the LDP is an 8086 assembler, linker, and debugger.

VMS is a powerful general-purpose operating system that provides a complete set of user programs and a rich collection of system services available to the programmer. The VAX Fortran language is an optimized and enhanced implementation of the Fortran 77 language, and is the standard for user level programming in VMS. A powerful debugger is provided to aid development. In

---

4 The author is not convinced that this is the direct cause of the problem, but he has been unable to run more than four processes concurrently. In his opinion, the problem is more likely to reside in the actions of particular processes that cause unusual conditions to occur inside the kernel.
addition, the VMS command language, DCL, can be used for writing batch commands.

An simple implementation of Zephyr exists on the VAX. To distinguish this from the Zephyr implementation on the LDP, this will be known as VAX Zephyr. This implementation of Zephyr consists solely of the kernel and a few simple utilities; the powerful utility programs on the LDP do not exist in VAX Zephyr. The term VAX Zephyr will normally be used to refer to the kernel without the utilities.

The GEMPAK graphical data analysis package on the VAX is an important resource for the CMPO's instructional and research program. It processes meteorological data in LDP format. The file transfer capabilities of SWTP increase the utility of this package by permitting selective transfer of large data sets from the LDP to the VAX, rather than relying on the relatively small storage of real time data that can exist on the VAX. Using SWTP, it is possible to transfer only those data sets that are immediately needed to the VAX.

The LDP has software to organize the incoming DDS data and massage it into a form suitable for archiving. The VAX lacks this software, but has the hardware needed for archival. Both machines require DDS data, the LDP for inspection and the VAX for use by GEMPAK. Since the VAX has only a single line that can be used for acquisition of both DDS and archival data, the need for communications from the LDP to the VAX exists. SWTP attempts to meet this need.
Chapter 3

Basic Data Types

The data that Zephyr/SWTP handles can be grouped into two main categories, input and output data. The input data is the raw DDS data received over the telecommunications line. The output data is a mixture of raw DDS data, binary and text files from the LDP, and commands issued on the LDP that are to be issued on the VAX. Some of this output data is considered to be a continuous stream, and some of it is considered to be individual messages, or datagrams.

The DDS data consists of a stream of messages. Each message consists of related meteorological data in pure uppercase ASCII in addition to a few specific control characters. This data can be, for example, hourly surface weather observations from anywhere in North America, or a bulletin on a hurricane in the Atlantic Ocean, or anything else that the providers of DDS deem suitable for inclusion in a message. The messages also contain a header and a trailer. The header includes a delimiter to separate messages and code that identifies the type and origin of the data, in a fixed format. The trailer consists of an end of message delimiter. The header is used by Zephyr to determine the disposition of the message, but is not used by SWTP per se. In addition, some messages consist of records, and some of these records are further structured. Some Zephyr utilities understand these records and their internal structure, but none of the extensions to Zephyr concern themselves with this finer structure.

The output data consists of three basic types: raw DDS data, files on the LDP that are transferred to the VAX, and VMS commands issued on the LDP that are transferred to the VAX and executed. The first is considered a continuous
stream of data; the other two are treated as datagrams that are sent from the LDP to the VAX. The model used for SWTP provides for DDS data to be transferred as a continuous stream of data, broken by file transfers and remote command execution requests.

The DDS data is sent in pure form. This means that entire DDS messages are sent to the VAX from the LDP, which simplifies processing on the VAX, since VAX Zephyr already exists. Files that are sent to the VAX can consist of arbitrary 8 bit data; they are not restricted to ASCII characters, and they can be arbitrarily long. Commands that are executed on the VAX are assumed to consist entirely of ASCII text characters.
Chapter 4

Design of Zephyr/SWTP

The design of Zephyr/SWTP is intended to meet all current needs of the CMPO and to anticipate future needs. Two designs are presented in this chapter. The first design to be presented is the design that was finally implemented, and the second is an enhanced design that would be desirable to implement but could not be implemented in the time available for the project. The enhanced design shares many of the elements of the final design, and builds upon it to specify a much more sophisticated system.

4.1 The Final Design of Zephyr/SWTP

SWTP is based upon traditional network design. It treats the communications line as a very simple network, and provides the software to support the net. One important principle is modularity; the design specifies several more or less independent pieces of software which communicate through a simple, completely specified interface. In this case, the design has several hierarchical layers: a low-level layer that is actually responsible for putting data on the network, a middle layer that takes care of all bookkeeping, such as spooling, and a high-level layer that provides tools for the user to access network services. The lower layer roughly encompasses the data link and network layers, and part of the transport layer as defined by the Open Systems Interconnect (OSI) model of the International Standards Organization (ISO). The middle layer is equivalent to the

---

5 There is really not network layer in SWTP; network layers are only important when multiple computers have to share a network, which is not the case in the SWTP system.
rest of the transport layer and part of the session layer, and the highest layer corresponds to the rest of the session layer. The layers will be called respectively communication, data, and interface layers in the rest of this thesis.⁶

The communication layer is primarily responsible for sending data reliably and unambiguously to the VAX. Unambiguous means that the VAX knows what the LDP is trying to send; for example, this means that the LDP is actually transmitting real time data when the VAX thinks it is. Thus, the communication layer protocol must be correct in the sense that the VAX must always be at the same point in the protocol (the same “state”) as the LDP. This is done by providing signals to the VAX embedded in the data stream. Since there is one communications channel that only transmits eight bit characters, this should be done by means of eight bit characters.⁷ The communication protocol must take into account the fact that the user may wish to send any string of characters whatsoever, which may include character strings that are normally used for signalling.

The data layer provides an interface to Zephyr and the user for transmission of data messages or streams to the VAX. To VAX Zephyr it provides an input device that looks like a continuous stream of data; to Zephyr on the LDP it provides

---

⁶The OSI model of network systems architecture is an abstract standard designed for maximum network flexibility. In practice, few networks can be modeled completely by this standard, but it serves as a useful metric when discussing networks. Roughly speaking, the data link layer supports communication between “interface message processors” (IMP’s), which are themselves computers that take messages from host computers and pass them to other IMP’s, which pass the messages along to other hosts. The network layer is concerned with delivering messages from host to host. The transport layer provides a reliable communication channel between services residing on hosts, and defines a set of low level services that are used by higher level protocols in the session layer. The presentation layer, which is rarely formally implemented in practice, provides a high level interface to the session layer such as encryption of data or translation of incompatible character sets. The interested reader is referred to texts on networking; one such is listed in the bibliography.

⁷There are other ways of providing signals, such as by not transmitting for a certain period of time, but these can be ignored.
an output stream that is capable of continuously accepting data. To other programs it provides services that are capable of accepting an LDP filename and a VAX filename, and transferring the LDP file to the VAX file. It also provides a service that accepts a command string from the LDP and executes it on the VAX.

The interface layer presents a simple interface to the user or to batch command files written by the user. Thus, command files can be created that will run automatically at certain times of day that use simple interface layer commands that access the data layer services. We have little to say about the interface layer in Zephyr/SWTP, since it has changed very little from the interface software available in the old system.

The communication layer makes no distinction between file transfer and command execution; in both cases it sends a single message to the VAX. It does distinguish between these two services and stream data transfer. For message (datagram) transfer, the communication layer sends one message to the VAX and then goes off and does something else, such as sending another message. With stream transfer, on the other hand, the communication layer recognizes no termination of the data stream. Whenever it finds stream data ready to send, and is not sending something else, it sends this data.

One interesting wrinkle that the communication layer provides that many traditional networks do not provide is two priority levels. The communication layer may send datagrams at either high or low priority. High priority datagrams are always sent immediately; the DDS data stream is forcibly interrupted and the datagram is sent. Low priority datagrams wait for the DDS datastream to be momentarily empty before being sent to the VAX. The CMPO anticipates using this feature to allow the user to send commands and files at high priority, and to send archival data sets at low priority.
The communication layer always sends entire datagrams; it does not break up either datagrams or stream data into small units called packets as most conventional networks do. This is a weakness in the design; if a very long datagram is being sent the SWTP system may run out of queueing space, which would cause serious problems. There are other disadvantages to message based communications as opposed to packet based communications that will be discussed in Chapter 6 and in the section below discussing the enhanced design. However, a datagram may interrupt the data stream at any time. This is done by sending an interrupt sequence, which consists of a Control-Z (1A hexadecimal) character followed by the datagram. To permit insertion of Control-Z characters in transmitted data, they may be proceeded by a Control-V (16 hexadecimal), which is a quote character: any character may be preceded by a Control-V, which tells the communication layer not to look at that character, but to send it to the VAX. The Control-V must also be sent to the VAX so that the VAX knows not to try interpret the character as having a special meaning. Of course, SWTP must not attempt to force a datagram in by sending a Control-Z immediately after sending a Control-V but before sending the quoted character.

The communication layer on the VAX simply reads the incoming data and passes it to the data layer with no interpretation. It does not have to deal with the complexities of multiple priority levels. All that it really has to do is manage the output port and its input buffers.

The data layer accepts information from higher levels and passes it to the communication layer. This software provides a high level interface to various programs at the interface level, and shapes the data into a form that can be used by the communication layer. The data layer provides five services to the interface layer: file transfer and remote command execution at high and low priority, and
stream data transfer. These services are known by specific names; interface level programs can connect to the data layer by performing a system call with a particular name and the operating system will establish a connection. An example of the sort of operation that the interface layer would perform is to open a system queue with a given name.

For file transfer, the data layer accepts two filenames: a local filename and a VAX filename. It also accepts a flag indicating whether or not the local file should be deleted after it is transferred. The data layer opens the local file. If this succeeds, then it sends a header consisting of a single character (F for file transfer or C for remote command execution) to tell the communication layer on the VAX that a file is about to be received, then the VAX filename followed by the data in the file to the communication layer. When it has finished sending the data, it tells the communication layer that the session has ended by sending a Control-Z, and closes the local file, deleting it if necessary. If any characters in the file need to be quoted, the data layer arranges for this to happen.

Remote command execution is quite simple. When a command is to be executed on the VAX, the data layer sends a command execution header followed by the text of the command to be executed, quoting any characters that may need quoting. Stream transfer is also very simple; the data layer simply reads the data stream and sends it to the communication layer, quoting as necessary.

The most important service that the data layer provides is transparency. The interface layer sends the data layer very simple messages, and leave the details of reading files from disk, building headers, etc. to the data layer. This makes modification of the data layer a simple task, and does not require the interface layer to know any details about the data layer.
The data layer on the VAX is quite simple. It simply strips quotation characters from the data stream from the communication layer, determines what service is requested, and performs that service. By default, it assumes the incoming data is stream data bound for VAX Zephyr, and passes the data to the program. If file transfer is called for, the data layer opens the file and writes the data stream to the file. If command execution is called for, then the text of the command is passed to the VMS command interpreter.

The interface layer is very simple. It consists of a set of utilities that perform file and command execution for the user or for batch jobs. Zephyr itself is part of the interface layer, since it supplies a stream of data to the data layer. The interface layer does not exist on the VAX, since the LDP interface layer simply expresses high level requests in terms of specific operations to be performed on the VAX.

4.2 An Enhanced Design for Zephyr/SWTP

The most important difference between the design as implemented and the enhanced design is that the enhanced design of Zephyr/SWTP is more flexible and robust. It permits bidirectional communication, which could be used for additional services such as remote login. It also reduces the chance of data loss. This design is a high level outline, not a completely specified system as is the final design.

The communication layer will change the most, and its functionality will be split into a data link layer, which is responsible for reliable transfer of bytes between the two machines, and a higher layer which will be called the communication layer. The communication layer will use packet switching rather
than message switching. This means that messages, whether continuous stream or otherwise, are broken up into smaller units, called packets. In a typical packet switching network, each communications session gets to send a packet in round-robin fashion. Packets have a maximum length, so that each session will have a chance to send a packet without an excessive wait. This means that a complete session may last longer, but it also ensures that a session will not be held up for too long waiting for another session to complete. For example, a long file transfer will not prevent real time data from being transmitted, so the real time data stream is at no risk of filling its buffers.

A packet consists of a header and a body. The header tells the receiver which session the body of the packet should be delivered to. The receiver inspects this header, determines what session wants the data, and sends the body of the packet to the session. Thus, the session sees an unbroken stream of data, whereas in fact the data is broken up into small packets for communication purposes. Appendix C discusses the details of packet switching and the data link layer, along with some specific recommendations.

The communications layer can accept more than the three specific service connections discussed above. It starts out knowing about no connections, and creates connections based upon requests from the data layer. The only information that the communication layer needs is a name for the session and the priority of the session. The name can be anything that the data layer wishes to call it. This will be used by the data layer in the headers of all packets to be transmitted for that session, and for any other requests that the data layer may make about that particular session. The priority of a session can be changed by request from the data layer; the data layer supplies the name of the session and the new priority. The communication layer supplies the data layer with a count of how much data is
waiting to be sent to the VAX for each session, and how much data has been sent so far on this session.

The priority is used to determine which session will be allowed to send packets. The highest priority session with data waiting to be transmitted is selected, and a packet from its queue is sent. Sessions of this priority continue to transmit packets until they do not have enough data waiting to constitute a maximum size packet waiting to be sent. Sessions of equal priority transmit in round robin order. The next highest priority sessions then get a chance to transmit. After each packet is sent, the communication layer checks each higher priority session to determine if enough data to make a full size packet exists. If there is enough data, then that session gets a chance to transmit. If a session has its priority raised, then the new priority takes effect immediately.

There are four exceptions to the rule that complete packets always take priority over fragments:

1. If a fragment constitutes the end of a session (i.e. the session has been closed by the data layer, but some data remain to be transmitted), then this fragment is transmitted as if it were a complete packet, which is to say in normal priority order.

2. If no sessions have enough data waiting to constitute a complete packet, then the priority rules that apply to complete packets instead apply to fragments.

3. If a session has the highest possible priority, its data are sent immediately without waiting for a complete packet.

4. If a session has the lowest possible priority, then fragments of higher priority sessions take priority over full packets of this session. Fragments are never transmitted for minimum priority sessions unless they are the closing data for the session.

If a session has insufficient data to fill out a maximum size packet, but the session is closed, then the rule about full packets taking priority over small packets is suspended. This is important, as the character count of a session need not
consist of an integer multiple of the packet size. The session should not wait until all other connections' queues are empty before finishing. The purpose of the rule regarding transmission of fragments is to penalize sessions that transmit very slowly. If a session that generated output very slowly ran at high priority, it might force the communication layer to send a lot of small packets, which is inefficient. The last packet of a connection does not fall under this category, because it will have to be transmitted sooner or later, and it will not become larger by waiting.

The second exception is to ensure that if nothing else is using the channel, the channel bandwidth is not wasted. The purpose of the fragment rule is to minimize bandwidth loss due to packet headers, which are constant length, taking up space from packet bodies, which are not. In the case where no session has a complete packet ready, this does not apply, since sending some data is better than sending no data at all.

The third exception permits a session that consists of very short internal messages that are very critical to send without waiting for anything else, and allows sending data when nothing has enough data queued to permit sending full packets. A session that wishes to send at top priority should lower its priority and send whatever data it deems necessary to the communication layer. When it has finished sending the data, it should raise its priority to the maximum and wait until the communication layer has sent it all, then lower its priority before sending more data.

The fourth exception permits a session not to transmit until another session is finished. This is useful for, say, a remote command that depends on a file existing on the VAX. Normally, this sequencing could be assured by simply setting the command's priority lower than the file's priority. However, if for some reason the data for the file were slow in coming, a situation could arise in which a less than
a full packet's worth of data were ready for transfer. In that case, the command could be transferred to the VAX before the entire file had been sent. By setting the priority of the command transfer to minimum, it can be assured that the command will not be transferred to the VAX until the file is safely over. When the file session is closed, then the command's priority can be raised to just under that of the file transfer, since the only possible fragment of the file remaining will be the closing fragment, which takes priority over the lower priority command execution.

Checking all priorities after each packet is quite slow if there are a lot of connections. The task can be simplified by storing connections in a fast data structure, such as a heap. It is only really necessary to adjust the priority level immediately in the case of a connection whose priority, new or old, is either maximum or minimum, since special rules apply to these priority levels. Connections of intermediate priority normally need not be adjusted instantaneously; since their associated processes may run in any order, the exact order in which a connection's priority is adjusted is not important.

The ability to change priority on the fly is useful for Zephyr. If Zephyr knows that it is very backed up due to high priority file transfer, it can raise its priority until its queue has emptied out somewhat, at which point it can lower its priority. This ability prevents the lock up that is possible with the currently implemented protocol.

The data layer will gain new capabilities to match those of the communication layer, and will probably merge with the interface layer to some extent. The reason for this is that interface layer programs such as Zephyr and file transfer utilities will have to understand the dynamic priority adjustment facilities and the sent and unsent count in order to fully take advantage of the powerful communication layer. Some examples of uses of these facilities have already been outlined. Most importantly, the data layer will require a flexible naming scheme for
services, as well as priority specification. This will require closer cooperation with higher level programs. The details of the data layer have not been worked out.

The receiving software will also change considerably, but not to the extent that the transmission software will change. The communication layer and the session layer will separate to a greater degree than in the implemented system. Most importantly, the communication layer will have to route packets into specific data layer connections and recognize when connections close. In addition, since this design supports bidirectional communications, both the VAX and the LDP will have to support both the transmission and reception sides of the protocol.
Chapter 5

Implementation of Zephyr/SWTP

The implementation of Zephyr/SWTP closely follows the design presented in the previous chapter. It is not the cleanest possible implementation, but it was intended to be modularly separable from Zephyr for ease of modification. The points of contact between SWTP and Zephyr are few and well-defined. This implementation has only been partially tested; in particular, the communication layer has not been tested yet because certain hardware requirements for the system have not been met.

The amount of code written for the LDP was very much greater than the amount of code written for the VAX, which is not surprising in light of the design. For the VAX, one short program had to be written and one other program (VAX Zephyr) will have to be modified to take input from this program rather than from the actual port on the VAX. On the LDP, one program (Zephyr) required minor but significant modifications, several small utilities had to be replaced (the file transfer and remote command execution programs described in the design), and one program (Spooler, formerly PMON) had to be extensively modified. In addition, VMON, the old batch transfer program, was eliminated.

5.1 Implementation on the LDP

The code on the LDP makes extensive use of system queues. Five queues are maintained by the data layer implementation, one for each source of input. They are named REALTIMQ, BHIFILEQ, BLOFILEQ, BHICMNDQ, and BLOCMNDQ. The first queue is for real time data supplied by Zephyr, the others
are for datagram services: high and low priority file transfer and command execution services. The interface layer programs (Zephyr and the utility programs) communicate with the data layer program (Zephyr) through these queues.

The four utility programs provided are named VAXHI, VAXLO, VAXCMDH, and VAXCMDL. These perform the same functions as the old VAX and VAXCMD programs. The VAX programs transfer a single file on the LDP to the VAX, and the VAXCMD programs execute a single command remotely on the VAX. These programs simply open the appropriate system queue and put either the two filenames (VAX) or the command to be executed (VAXCMD) on the queue. Commands are terminated with an end of file (Control Z, 1A hexadecimal). Filenames are separated by a NULL (0 hexadecimal). The VMON program, which previously logged into the VAX, transferred data, and exited, has been eliminated, with its responsibilities assumed by the Spooler program. One consequence of this is that the /NOLOGIN option to previous VAX communication programs has been eliminated.

The spooling program, Spooler, was the most extensively modified program. This program was formerly called PMON because it provided spooling service to the line printer. Since it now provides data level service to other devices, it was renamed. The overall structure of Spooler was retained, but drivers for the high and low priority datagram and stream outputs were added. An interrupt handler, which serves as the communication layer, was written. It is assembled as part of Spooler and shares Spooler's address space, but is logically distinct and does not execute as part of Spooler.

Spooler is contained in SPOOLER.A86, and includes libraries SPOOLSUB.A86, SPSUB1.A86, and IOQ.A86. SPOOLER.A86 contains "generic" (non device-specific) initialization code and the main program loop, which handles
in order the line printer, high priority datagram, low priority datagram, and stream
devices. SPOOLSUB.A86 contains the low drivers for these devices. These
drivers are responsible for sending characters to the four devices. SPSUB1.A86
contains the entire communication layer interrupt handler, along with code to
initialize the serial port. IOQ.A86 contains definitions of the system queues used
by Spooler, along with a subroutine package for performing I/O operations on the
queues and initializing them.

The interrupt handler, in SPSUB1.A86, is the only piece of completely
original code on the LDP in this implementation of Zephyr/SWTP. It is very careful
to ensure that nothing will ever interrupt it, and takes precautions against some
unforeseen event somehow causing it to re-enter itself, since it is not re-entrant (it
uses static data storage). It is careful to ensure that interrupts are turned off at all
times when it is performing any operation with side effects, and in addition locks
itself against somehow being re-entered. This is accomplished by means of setting
a lock, which is implemented as the v_lock bit in vostat, which is the interrupt
handler's flag register. Immediately upon entrance, if the interrupt handler is
invoked somehow while another instance of itself is running, it will detect the set
lock and exit. The code is also careful to avoid any looping behavior to reduce the
chance of an infinite loop, which would lock the system up.

Following the setting of the lock, the interrupt handler sets up its segments,
saves its old stack pointer, and saves all registers that it will use. It also zeros a
counter named nothingsent, which is used by Spooler proper, as described below.
Note that saving the stack pointer and registers must be protected against re-
entrance, since the stack pointer is saved in a fixed place and the stack occupies a
fixed place in the interrupt handler's address space. After performing these
bookkeeping operations, the interrupt handler enters its body, at location vproc.
The VAX may not be able to keep pace with the continuous 9600 bps data stream that the LDP would like to send. Thus, it is necessary for the VAX to tell the LDP whether it can accept data. The process of regulating the data flow so as not to overwhelm the receiver is called flow control. The flow control in SWTP is implemented at vproc, where the interrupt handler first reads the input port of the VAX. If a Transmit Off (ASCII XOFF, or Control S) is found, then the interrupt handler sets the v_xoff flag in vostat, and returns. It does not attempt to transmit again until it receives a Transmit On (ASCII XON, or Control Q) from the VAX. This implements a flow control scheme known as XON/XOFF flow control.

After checking that the output port is indeed ready to accept data, the handler checks if it is currently transmitting high priority data; if so, it sends one byte of high priority data. If it is currently transmitting low priority data, then it sends one byte of low priority data. If it is sending real time data, it first checks that the previous character it sent was not a quote character; if it was, then it has to send a byte of real time data. If not, then it checks to see if there is any high priority data to be sent, since high priority data preempts real time data. If there is no high priority data to send, then it checks if any real time data needs to be sent. If so, then it sends one byte of real time data. If not, and the last real time character sent was not a quote character, it checks if any low priority data needs to be sent.

When the interrupt handler starts to send datagram (batch) data, it has to send an initialization character, which is Control Z (1A hexadecimal). After it sends this character, it sets the appropriate flag in vostat (either sthipri or stlopri) and returns. The next time the interrupt handler runs, the appropriate handler will be given control. It sends one byte from the appropriate buffer, and sees if it is sending an unquoted Control Z. When it sends this character, it resets vostat to
strtime, indicating that the handler should consider itself to be sending real time data. The next time the handler runs, it will check the high priority, real time, and low priority queues in order.

In addition to nothingsent, the interrupt handler sets another counter for the benefit of Spooler, named vintsent. This is a 16-bit count of the number of characters the interrupt handler has sent to the VAX since it started. It will wrap around after 64K characters, but this is not important since the total buffer space allocated is 12K bytes, and vintsent is compared to the number of characters that have been buffered for the interrupt handler. The difference between these numbers cannot exceed 12K. This is used by spooler to determine how far behind, if any, the interrupt handler is. Its precise function will be described below.

The interrupt handler is not paranoid about keeping the state of its buffers completely correct at all times; rather, it is careful that the pointers into the buffers are always within one byte of their correct location, and never contain any completely garbaged value. The combination of turning interrupts off and locking the interrupt handler should be sufficient to prevent any problems, especially since the interrupt handler never executes any code outside its body. At worst, a duplicate character could be sent, which is only fatal if it is either a Control-Z or a Control-V. However, the interrupt handler should be completely immune to re-entrance.

The drivers in SPOOLSUB.A86, however, must be much more careful never to leave their pointers garbaged, since the interrupt handler checks for a difference between the insertion pointer and removal pointer to determine whether useful characters exist in the buffer. They are careful to insert the character into the buffer before incrementing the insertion pointer, so at worst there will be a character in the buffer that the interrupt handler does not see. This is not
important, since Spooler is never allowed to completely fill the buffer. More importantly, when the circular buffers wrap around, the value stored in the insertion pointer is always legal. By turning off interrupts, the drivers can be certain that the insertion pointers will be atomically updated. Vlocalsent, a count of how many characters have been sent to the interrupt handler’s buffers, is updated in the same atomic operation.

Spooler itself runs in a loop starting at the label “top”, servicing the printer and the three logical output channels to the VAX, then dispatching (relinquishing control of the CPU) to permit other processes to run. In one cycle of the loop, it processes one character for the printer and up to 128 bytes (one disk sector) for each of the three VAX outputs. In addition, it checks that the interrupt handler is running regularly. This is important because the serial port only sends an interrupt when it is first ready to transmit a character. If the interrupt handler does not transmit a character at this time, it will not be triggered again unless something else, such as Spooler, triggers it. Spooler uses the counters nothingsent, vlocalsent, and vintsent to determine what action it should take.

If vlocalsent is equal to vintsent, then no data is buffered for the interrupt handler. In this case, there is no reason to take any action, since the interrupt handler will not be able to send any data. If there is data waiting to be sent, then the counter nothingsent is checked. Nothingsent counts the number of times that Spooler’s main loop has run without the interrupt handler being invoked and making it past the locking stage. If nothingsent is 5 or less, then no action is taken. If nothingsent is greater than 5 but less than 10, then the v_lock bit of vostat is checked. If the interrupt handler is unlocked, than a software interrupt is generated. If nothingsent is 10 or greater, than the interrupt handler is assumed to be stuck for some reason. The interrupt handler is unlocked forcibly, and an
interrupt is generated. This may cause transmission of a duplicate character if the interrupt handler somehow crashed between sending the character and incrementing its pointers. Following the interrupt, nothingsent is reset to zero. The purpose of this exercise is to ensure that an interrupt is generated for the interrupt handler if necessary, but that no software trap is generated otherwise.

The processing of both VAX datagram queues is identical. The strategy employed is to read one 128-byte message from a system queue and to insert the complete message in the buffer for the interrupt handler. Since theoretically every character in the message could require quoting, and since it is essential that the insertion pointer never equal the removal pointer to the buffer when there is live data in the buffer, we require that at least 257 bytes be available in the buffer. This is not a serious restriction because each buffer is 4K bytes long. The amount of memory wasted is insignificant.

We will discuss the code for the high priority queues, since the code for the low priority queues is line for line equivalent. Following the check for buffer space, the mode flag vhimode is checked. Vhimode is one of FILE, RT, or NEITHER, corresponding to file transfer, remote command transfer, or neither taking place. If vhimode is FILE or RT, then the appropriate queue is processed. If it is NEITHER, then first the command queue, and then the file queue is checked for data. This order is arbitrary.

If data is detected on the file queue, then one message is read from the file queue and the first eleven bytes are treated as a filename on the LDP. The next byte is a delete flag; if it is non-zero, the local file will be deleted after it is transferred. If the local file is opened successfully, then the character 'F' is sent to the high priority queue, which will tell the VAX that a file transfer operation follows. The VAX filename is read from the remainder of the message and from any further
messages until a NULL character is found. The VAX filename is not quoted; it is assumed to consist solely of ASCII printing characters. The file is then read from disk one sector at a time and the data sent directly to the high priority queue, quoting all control characters with Control-V. When the end of file is reached, an unquoted Control-Z is sent, the file is closed and deleted if necessary, and vhimode is reset to NEITHER.

Commands are processed similarly, except of course no file has to be opened. The header for commands is the character 'C'. Nothing is quoted by the command processing code at vhi_cmd_monitor, however. This is because the only way to know when the end of a command has been reached is for the interface layer to tell the data layer. The only way for the interface layer to indicate that a command has finished is to send an unquoted Control Z. Thus, if the interface layer wants to send a Control Z, it must perform the quoting itself. For the data layer to quote anything would thus be pointless. The interface layer must perform any quoting.

The handler for the real time queue is much simpler, since it never has to look for a terminator. It simply reads one message from REALTIMQ, quotes as needed, and sends the data to the interrupt handler.

Zephyr was modified in five places, two of which pertain to the operation of the program and are visible to the user, the other three of which are communication related. The first place was in the code to read in the control file. Zephyr's control file specifies what should be done with each message type; currently it specifies what file, if any, Zephyr should save the data in, how long it should keep the data, etc. The read in code in procedure addlst in file NZEPHYR.A86 has a few lines added to store a flag indicating whether data should be sent to the VAX. The flag should be ASCII '0' (30 hexadecimal) if data is not to be sent to the VAX and some
other value (typically ASCII '1') if data is to be sent to the VAX. The second place that Zephyr was modified is in procedure getnam in NZEPHYR.A86. Near the tag reject, the code checks the VAX flag for the particular data type and sets Zephyr's VAX status flag, called vaxstat, to the value called for in the table.

The third place that Zephyr was modified is between getch1 and getch3 in NZEPHYR.A86. This code, in the area where data is removed from Zephyr's input queue, checks first that there is no error condition, then checks that the character received is a meaningful Zephyr character and places it in the VAX buffer, named vaxbuf. If the VAX buffer is full, it calls the new procedure flushv, which is the fourth modification. Flushv is also called if a Start Of Header (ASCII SOH, or 01 hexadecimal) character is found. This is important, as described below.

Flushv is an important procedure, and was implemented in such a way that very little code in Zephyr needed to be changed. This function checks vaxstat to determine whether or not to send the buffer to the data layer. Data is always buffered for the VAX in the getch code, but is not sent to the VAX unless vaxstat is on. If data is sent to the data layer, it is sent as a message on REALTIMQ.

There are several special cases that flushv has to deal with:
- When Zephyr receives a message header, it cannot turn on vaxstat until it has received the entire message header. Flushv has to arrange not to lose part of a header.
- Flushv can sometimes send an incomplete buffer. The message size of REALTIMQ is 128 bytes, but it is possible for smaller buffers to be sent.
- It is possible for an error to occur when writing to REALTIMQ. Flushv cannot wait until REALTIMQ can receive data and write the data, since Zephyr is continuously receiving data and must not block.

The solutions to all of these problems are fairly simple. The way that flushv always succeeds in sending a message header is by arranging for the first character of the header to be the first character in the buffer. The header line on a
message is never 128 bytes long, so Zephyr has already set or reset vaxstat before flushv is called to flush the buffer. This is why flushv is called immediately upon receipt of an SOH, before the SOH is placed in the buffer; it ensures that REALTIMQ's buffer will be empty when the next message header is received. It also ensures that no garbage from a previous message will be sent out along with a header.

This is why flushv has to be prepared to send a partially full buffer. However, with some cooperation from the data layer, to be explained below, this can be done easily. Flushv simply sticks a NULL character (0 hexadecimal) onto the end of a partially full buffer. The data layer then knows that anything after the NULL is garbage, and reads the next message from REALTIMQ. If Zephyr wished to send a NULL character, it would have to quote it. However, this situation does not occur, since NULL is not a defined character in the character set used by DDS.

Currently, the error recovery used by flushv can best be described as a "panic." If flushv is unable to write to REALTIMQ, it fills its buffer with an ETX character and a NULL. This sequence effectively closes off the current DDS message, if any, for the benefit of VAX Zephyr. It then sets an error flag, vaxerr, and turns off vaxstat. Before the getch code actually puts the character in REALTIMQ's buffer, it checks this flag. If the flag is set, it calls flushv, which attempts to send out this emergency message. If flushv succeeds, the error condition is cleared. Vaxstat can then be turned on when the next message header comes through.

The final modification is in the initialization code, in nzsubs2. The code, just before setmode, attempts to open REALTIMQ. If this open succeeds, the queue will be used by Zephyr/SWTP to send data to the VAX. If this open fails, Zephyr assumes that SWTP is not to be used this session and sets a flag to ensure that
the code around getch that handles the VAX will be skipped, effectively turning off the SWTP enhancements.

5.2 Implementation on the VAX

The implementation of the VAX (receiving) end of SWTP is far simpler than the LDP end. One simple program had to be written and one program (VAX Zephyr) required slight modifications. The new program simply reads the VAX input port and sends the data to either the VMS command interpreter, a file, or to Zephyr, strictly following the protocol. Zephyr was modified to accept input from this program rather than from the input port; the only thing to change was the initialization code.
Chapter 6

Analysis of Zephyr/SWTP

It is impossible to analyze the ultimate success of Zephyr/SWTP since the implementation cannot be completely tested due to lack of Transmit Ready interrupts on the serial port. Since it was impossible to test the system, and observe it in operation, no conclusions can be drawn as to its performance and suitability for use. Zephyr/SWTP was designed to meet the needs of the CMPO, but how well the actual implementation will work could not be determined as of this writing.

Some problems were noted during testing. The Spooler uses quite a lot of CPU power, even when quiescent (transmitting no data). The only way to work around this is to increase Spooler’s efficiency and have it check its queues less frequently. This may require increasing the amount of data read from disk during each file transfer cycle to keep up with the serial port.

The implementation of Zephyr/SWTP is simple and modular, but it has shortcomings. The most important problem with the implementation of SWTP is that it is hard to maintain, since in many cases large pieces of code are duplicated. This problem exists in three places in Spooler: in the main loop, in the drivers in SPOOLSUB.A86, and in the interrupt handler, where the code for handling low priority data is line-for-line identical to code for handling high priority data. This makes these sections of code difficult to modify, since any change to the system requires changing two locations in the code. A much better implementation would use a pointer to a data structure representing a logical connection. Then in order to send high priority data a routine would be passed a pointer to the connection structure for high priority data, for example.
There is very little explicit error handling in Zephyr/SWTP, with more emphasis on error avoidance. Programming so as to avoid errors in the first place is a good idea, but the first rule of systems programming is that errors that should never occur will in fact occur. In fact, the only real error handling, apart from checking that files are opened properly and the like, is in the main loop in Spooler and in the communication level interrupt handler. Spooler's main loop watches the status of the interrupt handler to reset it in case of lockup, and the interrupt handler locks itself despite turning off interrupts. There are several classes of errors that can be handled within the final design of SWTP but are not handled by the actual implementation.

The most likely error to occur, in our opinion, is that system queues will fill up. This is not a problem for file transfer or remote command execution, but is a very serious problem for Zephyr. Zephyr has no ability to cache data either in RAM or on a disk file, so it has to abort the current transmission when it runs out of space. The amount of storage on REALTIMQ should thus be made as large as possible. It should be possible to configure 128K bytes for this system queue, which at 1800 bps will take at least 12 minutes to fill, assuming reasonably long messages (very short messages will result in the transmission of frequent short system queue messages). This is a significant constraint, since it is common for large amounts of data that are to be transmitted to the VAX to arrive in a short period.

If the transmission line runs continuously at 19,200 bps, it can transmit data ten times as fast as Zephyr receives data. This would permit transfer of 1M byte of files or commands at high priority before Zephyr is in any danger of overrunning the system queue. However, the transmission line may be unable to maintain 19,200 bps due to conditions on the VAX, which are presently unknown. For example, the
VAX may be completely incapable of accepting data at 9600 bps for any length of time when GEMPAK is being used. There is no way short of testing the system under normal and worst-case usage conditions of determining what data rate will actually be maintainable. Table 6-l lists some examples of the size files in bytes that can be transmitted given specified line speeds, but these should not be considered realizable in practice. Table 6-ll shows how long in seconds it will take the communication layer to empty the real time buffer at these speeds, assuming continuous inflow of data; again, these are best case times. This was one motivation for designing the priority system, since archival runs typically transmit large blocks of data. It is not necessary for these files to be transferred immediately. While this won't protect against the transmission of a single large file, it will protect against transmission of a lot of files that are in toto very large.

<table>
<thead>
<tr>
<th>Data Rate (bps)</th>
<th>64</th>
<th>96</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>1800</td>
<td>64</td>
<td>96</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>2400</td>
<td>85</td>
<td>128</td>
<td>171</td>
<td>341</td>
<td>683</td>
</tr>
<tr>
<td>3600</td>
<td>128</td>
<td>192</td>
<td>256</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>4800</td>
<td>160</td>
<td>240</td>
<td>320</td>
<td>640</td>
<td>1280</td>
</tr>
<tr>
<td>9600</td>
<td>320</td>
<td>480</td>
<td>640</td>
<td>1280</td>
<td>2560</td>
</tr>
<tr>
<td>19200</td>
<td>640</td>
<td>960</td>
<td>1280</td>
<td>2560</td>
<td>5120</td>
</tr>
<tr>
<td>38400</td>
<td>1280</td>
<td>1920</td>
<td>2560</td>
<td>5120</td>
<td>10240</td>
</tr>
</tbody>
</table>

Table 6-I: Maximum File Transfers at Selected Communication Rates and Queueing Capacities (K bytes)

Another possible condition that is not handled well is the loss of an XON character from the VAX. If the VAX sends an XON character, and it is somehow lost, then data will back up indefinitely. Unfortunately, it is not possible to prevent this from happening because conceivably the VAX could turn off reception for an
<table>
<thead>
<tr>
<th>Data Rate (bps)</th>
<th>Memory Available for Queueing (K bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>64</td>
</tr>
<tr>
<td>1800</td>
<td></td>
</tr>
<tr>
<td>2400</td>
<td></td>
</tr>
<tr>
<td>3600</td>
<td></td>
</tr>
<tr>
<td>4800</td>
<td></td>
</tr>
<tr>
<td>9600</td>
<td></td>
</tr>
<tr>
<td>19200</td>
<td></td>
</tr>
<tr>
<td>38400</td>
<td></td>
</tr>
</tbody>
</table>

Table 6-II: Time Required for Queue to Empty at Selected Communication Rates and Queue Lengths (seconds)

arbitrarily long time. It may be necessary to simply trust the communications channel, and if a problem occurs, to analyze the particular problem and arrive at a workaround. The most obvious workaround is for Spooler to time out and turn transmission back on, but it is not at all obvious how to protect against the VAX truly being unable to accept transmission. The VAX could perhaps decide that if it has not received any data in a certain period of time, perhaps five seconds, that the LDP has not received an XON. If it sent an XON at this time, it would not do any harm, since if the LDP simply had nothing to transmit, it would quietly ignore the XON.

If modification of Zephyr/SWTP is attempted, fixing these problems is a good place to start. These two possibilities may never cause any problems, but they are two identified points in the whole system that may cause fatal errors. There are other ideas in the enhanced system design that could be implemented in a future version of Zephyr/SWTP.

It will be necessary to convert SWTP to a packet protocol from the current
message protocol for almost any extension. The simplest way to do this is for the communication layer to send out data in 128 byte units, each with a header. The header need only be a single byte indicating whether the data is to go to VAX Zephyr (i.e. is from the data stream) or whether the data is to go to a file or command execution (i.e. is from a datagram source, either the high or low priority data source). It may be desirable to send characters between packets that synchronize the VAX to the start of the packet. For example, it is impossible for the protocol to send three consecutive Control Z characters normally. A sequence of three consecutive Control Z's could indicate that the next packet begins after the last Control Z.

To fix the overflow problem, then, it is simply necessary to ensure that real time data can flow from the LDP as fast as it is received from the DDS. This can be done by reserving, for example, every third packet slot for packets from the stream data source when there is more than 1K byte of data in the communication layer's stream buffer. This minimizes the number of small packets sent, which increases the effective bandwidth. At the same time, it ensures that the DDS data stream gets enough bandwidth while letting the high priority data get most of the bandwidth.

At this stage, the differences between stream and message communications will be minimal. Both will be sent as groups of packets, and the only major difference between the two types of connections is that the stream connection will never be closed. In essence, then, there will be three priority levels. The high priority data will get most of the packets, the middle priority will be guaranteed a certain fraction of the packets, and the low priority will get anything left over.

The next stage of modification is improving the flow control mechanism. The details of this are discussed in Appendix C. A simple way of reducing exposure to
loss of an XON character is for the LDP to probe the VAX periodically when the LDP thinks that transmission is off. This can be done by requiring that XOFF characters in the datastream be quoted and sending three consecutive XOFF characters. The VAX is then required to send an XOFF character if it is ready to receive. One problem with this is that the VAX may not be ready to receive data until one or more of the XOFF characters has been sent, thus confusing the protocol. The workaround is for the VAX to send an XOFF while it still has buffer space for perhaps 8 characters, to ensure that it can receive a few more characters before the LDP receives the XOFF, along with any signalling characters.

Following this, the fixed connections to the data layer should be changed into flexible connections. This can be done with a queue of requests for connections (RFC’s) and a queue of replies. Each RFC consists of a name and priority. The name is the name of the system queue that will be used for communication with the interface level process and the data layer, and the priority will be a small number (perhaps 0, 1, or 2 using the scheme described above). When an interface level process wishes to open a channel, it locks the RFC queue for writing, and sends a message consisting of the RFC. The interface layer creates the queue, and if this operation succeeds it returns an acknowledgment (ACK) or a negative acknowledgment (NAK) if the operation fails or a queue of that name already exists. When the interface process receives the acknowledgment, it unlocks the RFC queue and opens the newly created queue.

The data layer must also create a connection to the communication layer. This consists of a buffer, a priority level, and a set of pointers into the buffer. A round robin scheme for servicing connections can be used by the communication layer.

The final step is to upgrade to the full enhanced system, using the full priority
scheme discussed in that design section. It may be desirable to add duplex communications at this point, also, which requires that both computers implement both sides of the protocol.
Chapter 7

Summary

The combination of Zephyr and SWTP provides a system that can be used by the CMPO for continuous transmission of real time weather data from the DDS, transfer of disk files to the VAX from the LDP, and remote execution of commands issued on the LDP on the VAX. These facilities should be sufficient to meet the needs of the CMPO. If additional facilities are desired in the future, they can be added in a stepwise fashion to create a system that should meet any needs of the CMPO as long as the LDP and the VAX together are sufficient.

SWTP itself is a simple network protocol that supports the three above mentioned services. It was installed into the original Zephyr system with minimal alterations to existing code. This means that the SWTP enhancements should have minimal impact on the well-tested Zephyr software. SWTP itself is modular in that the separate layers of the software can be modified and even re-designed with little or no impact on the rest of the networking software. While we can have no confidence in untested code, such as the interrupt handler, we believe that the system as designed will meet the needs of the CMPO with regards communication capabilities from the LDP to the VAX.
Appendix A
Non-Interrupt Driven Transport Layer

At the time this is being written, it is not completely certain that interrupts will be available on the serial port connected to the VAX. Here we discuss the problems that will arise and provide some ideas on how to cope with them.

First of all, it should be recognized that interrupts make the system far more efficient than it would otherwise be. The only alternative to interrupt driven output is polling of some sort. Polling requires that the CPU spend a large fraction of its cycles watching the I/O port. These cycles cannot do anything “useful”, so polling slows the computer down considerably. This will adversely impact upon the performance of the Zephyr utilities, which will be quite noticeable to the user.

The old PMON program manages to handle the printer without any trouble without explicit interrupt driven I/O. However, this only works due to a quirk of fate. The printer nominally runs at 1200 bps, but in fact is capable of accepting just over 100 cps, and clock interrupts happen at a rate of 100 per second. Thus, PMON can feed characters to the printer at its maximum rate if it runs every clock cycle, or nearly so. To ensure this, it sends one character, and causes a dispatch, allowing some other process to use the CPU. If the load on the system is low, as it normally is, it regains control of the CPU on the next clock interrupt. PMON is therefore very much an interrupt driven program, even if the interrupts are coming from the system clock rather than from the I/O channel.

This technique will not work with SWTP. The Zephyr input line runs at 1800
bps\(^8\), which means that data is entering the system faster than it is being drained\(^9\). In addition, once each day archival runs are made to the VAX, which typically transfer on the order of 200K bytes of data or more, and a general file transfer facility is envisioned.

The output line to the VAX runs at speeds up to 19,200 bps, which is easily sufficient for most normal applications, and with improvements in the transport layer, could handle just about anything that anyone wanted to throw at it, if the line is allowed to run at full speed. Therefore relying upon clock interrupts alone will not work. In addition, it is not desirable for the system to use every clock interrupt, since this will prevent other applications from running. Making the system transparent to the user is an important design goal; if the system is unusable during periods of high data transfer rate, then this goal is certainly not satisfied.

It is important to maintain a high enough transfer rate to ensure that when DDS data is backed up due to the transfer of files to the VAX, the backlog will be cleaned out in a reasonable amount of time. A mean transfer rate of at least 3600 bps (360 cps) averaged over a five minute period is probably necessary to achieve adequate performance. This will ensure that backed up real time data clears out at a reasonable rate. The short term average data rate should be somewhat higher to allow for disk waits during file transfer.

Concurrent DOS allows transfer of one or more 128-byte disk sectors per

\(^8\)Future communications lines will run at higher speeds, perhaps as high as 4800 bps.

\(^9\)While this is a worst case scenario, the worst case is also a common case; once each hour for roughly ten minutes data is flowing into the system continuously that will be sent to the VAX. Twice each day, an even greater volume of data needs to be sent to the VAX.
operation. At 19,200 bps (roughly 2000 cps), transfer of this data to the output port will take on the order of 60 ms. Assuming that a disk transfer takes 75 ms, if we must use synchronous I/O, we are limited to a maximum bandwidth of roughly 6000 bps; allowing for worse cases and the fact that the spooler will not immediately regain control when the disk access is complete brings this number down somewhat; it is unlikely that performance much better than 8000 bps, if even that, can be achieved under these conditions. This is comfortably above the requirements of 4800 bps burst and 3600 bps sustained performance for the current system, but it will not be adequate for future high speed systems. By performing multiple-sector reads, the transfer time can be reduced because reads of multiple sectors are almost always much more efficient than reads of a single sector.

The most obvious way to achieve this level of performance is for the spooler to simply set its priority very high to ensure that it gets every tick, and then to sit in a polling loop on the output port, sending a byte whenever possible. This will work, but during periods of high activity, it will prevent other applications from running. This is a serious problem, as Zephyr must get cycles, and the user also must be able to use the computer. If the spooler takes half of the available cycles, or takes half of each cycle, allowing other programs to take the other half, then the computer can probably run fast enough to perform all its tasks, although it will feel sluggish to the user.

Taking one-half of the total CPU time will work whenever no disk accesses are being performed, since in this case an average data rate of 9600 bps can

---

10 Concurrent DOS caches multiple logical sectors by reading a single physical sector. This fact will increase performance but it is not clear to just what extent.
perhaps be maintained, which is easily enough when there is only moderate backlog. And, of course, if there is no backlog at all then only 10 to 15 percent of all cycles will be needed, since if no data needs to be sent, then the spooler can relinquish control sooner. Fifteen percent of the CPU will be an almost unnoticeable load. 11

The problem is when large file transfers are being performed, particularly when there is a backlog of real time data. If the spooler is taking one half of the cycles by a simple minded scheme of taking one cycle and immediately relinquishing the next (which is very difficult for user programs to do) or by sending five characters and relinquishing control (which is easy to do), then the effective data rate will be one half of 8000 bps, or 4000 bps, which is dangerously close to the floor for effective operation. And since data is being output synchronously, the spooler can’t steal cycles during disk accesses, although other programs can use them. And these situations are the ones in which maximum bandwidth is most necessary.

Two solutions to this problem are presented here. They are called adaptive scheduling and multi-layered spooling. The purpose of adaptive scheduling is to ensure that the output port driver gets enough cycles to quickly clear a backlog when the backlog is large, while granting the rest of the processes that wish to run enough cycles when the load is lower. Multi-layered spooling ensures that the output driver is not slowed down by having to wait for disk accesses, which ensures that the output driver is really sending as many characters as it thinks it is sending.

11 There is empirical evidence that the spooler is a considerable load even when no data is being transferred. Rewriting the spooler in more efficient code will reduce the load.
The premise behind adaptive scheduling is that we can control the amount of CPU usage during non-disk access states by transmitting a fixed number of characters, then relinquishing the CPU. In theory, we can transmit almost 20 characters per tick; a more conservative estimate is that we could in fact transmit 15, allowing for overhead of both the operating system and the spooler. This number is important; if we try to transmit more characters than this we will spill over into a second tick. If we overflow into a second tick and dispatch, we will lose the rest of the second tick. This technique relies upon estimating how long it will have to wait for its next chance to run, so it will be confused if it overruns.

The number of characters to be transmitted before forcing a dispatch is a variable that can be set by the spooler depending upon its estimation of the load. Thus, under conditions of normal load, the spooler would send five characters before doing a dispatch. When it does disk accesses, it would probably want to use the entire cycle or very close to it; thus it would send perhaps 15 characters before dispatching. The numbers would probably be assembled into the program, and would be dependent on the precise length of a clock tick. As a result, the program would not be portable to different hardware.

Various modifications might be used. For example, while performing disk accesses, the spooler might not raise its “priority” until it had sent perhaps 4K bytes of data. And while flushing backlogged data, it might not reduce its priority until it had sent out one half of the total number of characters it had sent out during the prior period of disk accesses. Starting with the simple system first is probably the correct thing to do. If performance is inadequate, the more elaborate modifications can be implemented.

If we can eliminate the slow disk accesses, we can boost performance because we know exactly how many ticks we are getting; namely, almost all of
them. To do this we require an external process to spool data to the spooler. The external process merely reads the data from the disk and queues it to the spooler. This is the concept of multi-layered spooling.

This can be done with a variety of techniques. The simplest is for the file transfer programs to read the disk file and put the live data on the queue, rather than simply sending the filename. Another alternative is for the spooler to tell the disk reader what filename to read, by means of a system queue. If the two processes live in the same address space, then communication is easy and fast (e.g. buffers rather than system queues can be used for communication). By sending five or six characters before dispatch, we can maintain a communication rate of roughly 4500 to 5500bps, depending upon exactly how much CPU the disk accesses really take for the operating system.

The techniques of adaptive scheduling and multi-layered spooling can be combined very easily, since they are independent. One way is for the spooler to see how much buffered data needs to be sent out and to adjust its priority upward if too much data is waiting. A simpler way is to count how many queues need service as a means of checking load; if two file queues, a command queue, and the real time queue are each backed up, then the spooler probably wants to send out the full eight characters before dispatching. On the other hand, if only the real time queue or only one file queue need processing, then reducing the number of characters to five might work. Experimentation with real system use will be necessary to determine the appropriate parameters.

Interrupt driven I/O vastly increases the amount of data that can be sent to the VAX, while minimizing system load. However, the fact that interrupts might not be available is not necessarily fatal to Zephyr/SWTP. The system will run slower, and the rate of data transfer will be lower, but by means of careful design and clean implementation, it should still be possible to create a usable system.
Appendix B
A Testing Strategy for Zephyr/SWTP

The hardware was not ready for complete testing and debugging of SWTP by the due date of this thesis. Some of the higher level code, notably the interface and data layers of the LDP, can be tested without the communications and therefore were tested. The low level code on the LDP, and most of the code on the VAX, cannot be effectively tested without proper communications between the two machines. The communication layer on the LDP was written in such a way that it never loops, which reduces the chance of a fatal error that would hang the system. The VAX code is very simple, and is written in a high level language (Fortran); it should be easy to test.

The various subsystems of Zephyr/SWTP on the LDP can be tested independently, and should probably be tested in a top-down fashion. The high level portions of the system (the utility programs, the extensions to Zephyr, and Spooler) can be tested and debugged independently of the communication layer (interrupt handler). The utilities themselves can be tested to some degree by inspection, since they are very simple. However, the rest of the system is complex enough so that it will need to be tested more systematically.

First of all, the protocol itself should be verified. What the utilities and Zephyr emit should be what Spooler wants, and what Spooler wants should be what the interrupt handler expects. The operation of the Spooler has been verified completely, as have the operations of the interface level utilities. The interface between the Spooler and the interrupt handler is not verified, but the Spooler is outputting data properly, according to the system design.
The next stage is testing the interaction of the utilities with Spooler. The simplest way of doing this is for Spooler to write data into files rather than into circular buffers. The files should have names that reflect which buffer the data is supposed to be written to. It is important to check that quoting is being performed correctly by Spooler. This has also been done, and appears to work properly.

When this works correctly for all combinations of priority level and connection type (file transfer or command execution), Zephyr should be brought into the picture. The Zephyr/SWTP communications protocol has been tested, but the combination of Zephyr and Spooler presently causes system crashes that are hard to debug. When any message is printed out, it is usually an operating system notice of an uninitialized interrupt.\textsuperscript{12} It is possible that the testing technique (writing data into disk files) is in part responsible for the crashes, since the load on the disk is extremely high when the new version of Zephyr is running.

The interrupt handler cannot be tested until something on the VAX can receive the data. It would be possible to verify that the interrupt handler does not crash, but the interrupt handler runs at such a low level that the act of testing via simulation is almost as likely to introduce new bugs as find old ones. We suggest initially running a stub on the VAX that simply reads data one byte at a time and routes it into a file, the contents of which can be examined. The danger involved with the interrupt handler is that it runs with interrupts turned off, and thus if it goes into a loop the only way out is by rebooting. Only coding the interrupt handler in such a way that the risk of an infinite loop is minimized can reduce this danger. The other danger is that the interrupt handler might have unforeseen side effects, perhaps trashing memory that it is not supposed to. This can only be determined

\textsuperscript{12} This is why the author believes that the number of processes alone is not responsible for system crashes, but rather the total load on the system, particularly by low level programs.
by careful re-inspection and testing. The interrupt handler should initially be fed only stream data. The high and low priority message facilities should only be used when it is clear that the interrupt handler is transmitting data properly.

There are some things that are very difficult to test. The code in Spooler to unlock the interrupt handler is very hard to test, since it is hard to simulate situations that will occur in reality, since those situations are not yet known. The "soft" prodding of the interrupt handler is testable, since the interrupt handler will need resetting whenever data transmission has temporarily ceased. The "hard" unlock and restart of the interrupt handler, on the other hand, should be essentially untestable because the interrupt handler should never get into a state in which it is wedged. The behavior of Zephyr when faced with a full queue would appear to fall under this category, but in practice is not hard to test. The technique used to test this behavior is to turn off the spooler, causing data to build up in the queue. When the spooler is turned back on, the initial output of Zephyr can be examined. This behavior has been tested and appears correct, but has not verified conclusively.

On the VAX end, testing will be easier because VMS handles most of the low level data capture and buffering and provides better facilities for debugging, and because the VAX end of SWTP is much simpler than the LDP end. The two things that require testing on the VAX are the input handler (communication and data layers) and the interface between the input handler and VAX Zephyr. VAX Zephyr will be essentially unchanged, but the input will be from a different source, which will make the I/O slightly different. The input handler will need more thorough testing. It is important to verify that the input handler actually routes data to where it should go, either to VAX Zephyr, the VMS command interpreter, or into a file, and that quotation is handled properly, meaning that quotation is being stripped correctly and that the input handler is not being confused and is not looping anywhere.
The trickiest pieces of code to test will be the low level code on the LDP, including the interrupt handler and the buffer drivers in Spooler. The code on the VAX should not be too difficult to test due to the higher level nature of the code, the powerful debugger available, and the general robustness of VMS. The high level code on the LDP has for the most part been tested, but there are still some mystifying problems that cause fatal system crashes. The interaction of the high level code with the output drivers may be at fault, since the driver for testing purposes, which consists of writing data to disk is much more complicated than the driver for the production system, which simply writes data into a shared buffer.
Appendix C
Low-Level Extensions to SWTP

This appendix discusses some low level concepts that are needed for extensions to SWTP, yet do not fit in comfortably elsewhere. These include details on packet switching, flow control, and signalling on the channel from the VAX to the LDP (the back channel).

C.1 Packet Format

When designing a packet switching protocol, it is important to devise a suitable format for the packets. Packets normally consist of a header and a body, perhaps with a trailer. The header normally contains information about the destination of the packet, the type of packet, the size of the packet, and anything else that the network designer considers appropriate. There are tradeoffs, though; a large header carries more information, but it wastes more network bandwidth and/or takes longer to transmit. The body of the packet is raw data that is not interpreted by the software transmitting the packet, but is merely passed to the higher levels of software. The trailer, if any, typically contains a checksum for error detection; some networks put checksums in the packet header.

There are two basic approaches to packet formatting: fixed length and variable length packets. Fixed length packets are simpler to manipulate, but are less efficient, while variable length packets require more sophisticated software, but are more efficient. The problem with fixed length packets is that a full packet must be sent even if only one byte of data must be transmitted. Thus, if 128 byte packets are used, and the user wishes to transmit one byte of data, then a full 128
bytes must be transmitted. On the other hand, if messages are uniform in length or normally very large, then a fixed length packet format can be more robust and easier to implement. Many networks use variable length packets with fixed length headers, but sophisticated protocols like TCP/IP\(^{13}\) use variable length headers as well as variable length packet contents.

What goes into the header depends upon the needs of the network. Typically, destination information is included in the packet header, which might include a host address. Since SWTP deals with a two-host network, there is no need for a destination host address. Other information included in packet headers might be the connection address on the source and destination machine, to identify which connection the packet is associated with. This is useful for SWTP, since typically multiple connections will be in use concurrently. For example, the primary Zephyr communications stream is a connection, thus all packets associated with this connection will have the name\(^{14}\) of the connection in the header.

Other information that might go in the header is the sequence number. To ensure that all packets are received and stored in the proper order, the packet header might include a sequence number, which tells the receiver which packets it has received. Also, if a garbled packet is received, the receiver may be able to figure out what packet it is and request a retransmission. Sequence numbers may also be used for flow control, which will be discussed below.

The packet format that SWTP uses can be quite simple, since there is no routing to consider, and since it is unlikely that users will want to add arbitrary

---

\(^{13}\)Transmission Control Protocol/Internet Protocol is the standard protocol suite used by the DARPA Internet and in many other networks.

\(^{14}\)A name need not be an ASCII string at all. It may be a number, for example. Frequently names have structure, particularly in complex, general purpose networks.
network services. Due to the limitations of both machines, particularly the LDP, it is likely that a few pre-defined services will exist, and if any more are desired, they will be written by hand and added as special cases. Figure C-1 gives an example format that might be used.

Whether fixed-length packets or variable length packets should be used is an open question. Figure C-1 assumes a variable length packet format, but a fixed length format could be arranged by removing the data length field. Most messages are likely to be either shorter than 128 bytes (e.g. remote command execution) or much longer than 1K bytes (e.g. file transfer or stream transfer). Testing and examination of the actual data transmitted is necessary to determine the preferred technique. It may well be that very few messages are shorter than a large multiple of 128 bytes, in which case inefficiency due to inability to deliver short packets is unimportant. On the other hand, it may turn out that a substantial portion of the system's bandwidth is being consumed by transmission of filler bytes, in which case ability to transfer short packets will be useful.

<table>
<thead>
<tr>
<th>Char</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Separator</td>
<td>^Z</td>
</tr>
<tr>
<td>2-9</td>
<td>Header</td>
<td>Connection ID, window size, packet type, etc.</td>
</tr>
<tr>
<td>10-11</td>
<td>Header Checksum</td>
<td>Checksum for packet header</td>
</tr>
<tr>
<td>12</td>
<td>Data Length</td>
<td>Length of packet data field</td>
</tr>
<tr>
<td>13-i</td>
<td>Data</td>
<td>Packet contents</td>
</tr>
<tr>
<td>i+1-i+2</td>
<td>Data Checksum</td>
<td>Checksum for data field</td>
</tr>
<tr>
<td>i+3</td>
<td>Terminator</td>
<td>^Z</td>
</tr>
</tbody>
</table>

Figure C-1: Sample SWTP Packet Format
C.2 Flow Control

Flow control is the intentional restriction of data flow to prevent the sender from sending data faster than the receiver can handle it. It is essential in virtually any communications channel, particularly for a high speed channel such as the one that SWTP uses. There are numerous flow control techniques in use, of which we will consider two. XON/XOFF flow control is a simple technique that requires the receiver to tell the sender when it is unable to accept more data, and then to notify the sender when it is again ready to accept data. Sliding window flow control is a sophisticated scheme that is used in packet switch networks. In this technique, the receiver tells the receiver in advance how much it may send, and the receiver cannot send more until the sender tells the receiver to go ahead.

XON/XOFF flow control is widely used by terminals that cannot accept continuous high speed data and by other simple serial communication lines. This protocol specifies that the receiving machine will send an XOFF character (^s) when it does not wish to receive any more data and an XON (^q) when it is willing to accept data again. This requires relatively little bandwidth on the back channel, as long as the receiver sends an XOFF early enough so it can accept the data being sent while the XOFF is being transmitted.

This technique is not transparent, which is a serious problem. The XON and XOFF characters are reserved and cannot be used for any other purpose. What is needed is something less disruptive to the character stream. It is possible get

---

15 The SWTP channel is unlikely to run faster than 19,200 bps, which is not considered terribly fast by contemporary standards. However, it is running at that speed continuously and over a channel that was designed for a terminal, which does not normally send continuous data at 9600 or 19,200 bps.

16 This hasn't stopped anyone from using this flow control method. That fact is no excuse for perpetuating this design.
around this problem by "escaping" the XON and XOFF characters. One way of doing this is to precede either of these characters by a quote character if they are considered part of the data stream. Duplicating the XON and XOFF characters is another way of accomplishing the same purpose. In this case, sending an XON character can be accomplished by sending two consecutive XON characters.

A problem with XON/XOFF flow control is that there may be a delay between the sending and receipt of the XOFF character. This is not likely to be a problem in the simple network that SWTP runs over, particularly if the receiver is able to send an XOFF before its buffers are full. However, even on this network it is possible for the VAX to request an XOFF to be transmitted, and for the LDP not to receive the character for several milliseconds depending upon the exact behavior of the serial interfaces on the two machines. This protocol is frequently implemented in hardware, which will not have this problem. Since it is implemented in software in SWTP, this has to be taken into account if possible.

Sliding window flow control specifies that the sender tell the receiver how much data it can accept, and that the sender cannot transmit more until the sender has accepted the first batch. For example, the receiver may initially be able to accept eight packets. If the sender transmits five packets, it may not transmit more than three more until the receiver has acknowledged some of the first five. However, the receiver may acknowledge any packets as soon as it has accepted them. The normal implementation of this specifies that the sender may send packets within a certain range of sequence numbers. When the sender safely receives the first $n$ packets in this range, it advances the range of packets it may receive, and sends acknowledgments back to the sender. When the sender receives these acknowledgments, it advances its range of legal packets. This is why it is called sliding window -- the sender has a window of packets that it may
send, and the receiver has a window of packets that it may receive.\textsuperscript{17} When the packets are acknowledged, the window is advanced (it "slides") into a new position. Sliding window flow control is a much more desirable technique on a packet switching network because it is more flexible.

\textbf{C.3 Signalling}

One important capability of a good communications system is the ability of one party to signal the other party. Signalling is an essential part of flow control mechanisms like the sliding window technique discussed above. This type of signal falls under the category of an acknowledgment. Signals can be also be used, for example, to determine if a host is up. This type of signal is frequently called a "ping".

An acknowledgment, or ACK is simply a statement by one party that it has received a transmission of the other party. For example, it may be desirable for the VAX to acknowledge receipt of all packets or messages from the LDP. The LDP waits until it receives an ACK before sending the next packet or message. This can be used to implement a simple sliding window flow control scheme, with a window size of one.

Related to the ACK is the negative acknowledgment, or NAK. A NAK states that one party expected to hear from the other, but did not hear the message or heard a garbled message. This could be used to implement error recovery; if the VAX receives a bad packet or message, it sends a NAK to the LDP to request retransmission. Use of NAK's permits more rapid response to errors, since the

\textsuperscript{17}Since the sender may not transmit any packets outside the window, the receiver simply ignores them.
sender doesn’t have to wait until it determines that the receiver failed to send an ACK.

A ping is simply a request for an ACK, to ascertain that the other party is indeed running and is not down. The utility of such a signal in the Zephyr/SWTP environment is unclear, but it might be used by the LDP in conjunction with transmission acknowledgments to determine whether it should keep waiting for an acknowledgment or give up and throw away its buffered data and close connections. The VAX might also use it to determine that it will not be receiving any data for a while, perhaps alerting the user at the console that the LDP has crashed.

C.4 Duplex Communications

Extensive use of Zephyr/SWTP might make desirable communications from the VAX to the LDP. For example, it might be desirable for the VAX to tell Zephyr to change its control file. Two way communication such as this is known as duplex communication. There are two varieties of duplex communication. Half duplex means that only one side sends at any time, and the other side waits for the first sender to finish. Full duplex permits both sides to send simultaneously. The nature of the communication line permits full duplex communications between the two machines, and the software to handle this should be no more complicated than the software for half duplex communications. It may in fact be simpler to implement, since there is no need to detect which side is sending and to wait for it to finish.

Flow control and signalling will be more complicated with duplex communications. XON/XOFF flow control could be used if these characters are
quoted, but sliding window flow control is preferable. In order to reduce the number of acknowledgment packets sent, a technique known as piggybacking can be used. This means that if someone wishes to acknowledge a packet when it has a packet to transmit, it includes the acknowledgment inside the packet. This acknowledgment might take one byte in the packet header, which is preferable to sending an explicit packet to acknowledge a reception.
References

[8086 80] Rector, Russell and George Alexy. 
*The 8086 Book.* 

[CPM 85] Digital Research Inc.  

[Tanenbaum 81] Tanenbaum, Andrew S.  
*Computer Networks.*  
A classic text on computer networks. This book discusses computer networks starting with topology, then progressing up through the ISO model using illustrative examples from contemporary network architectures. Knowledge of some basic computing concepts is assumed, but the book can be easily understood by a newcomer to the field.

[Zephyr 85] Neilley, Peter.  
The reference manual for the original Zephyr system.