Transporter a modem communication program for the Apple.

Michael A. Carr

Follow this and additional works at: https://preserve.lehigh.edu/etd

Part of the Computer Sciences Commons

Recommended Citation
https://preserve.lehigh.edu/etd/2327

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.
TRANSPORTER

A Modem Communication Program for the Apple II

by

Michael A. Carr

A Thesis
Presented to the Graduate Committee of Lehigh University in Candidacy for the Degree of Master of Science in Computing Science Department of Mathematics Lehigh University 1983
This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

May 12, 1983

Date

Professor in Charge

Chairman of Department
"If the pocket protector fits...wear it!"

-Jane Pali
Table of Contents

ABSTRACT 1

1. Introduction 2

2. General Program Structure 3

2.1 Library Units 4
2.2 Logical Division of Files 6

3. Major Data Structures 8

3.1 Environment Records 8
  3.1.1 Apple Environment 9
  3.1.2 Host Computer Environment 12
3.2 Altering Absolute Memory- the Free Union Record 14

4. Modes of Operation 20

4.1 Creation and Modification of Parameter Files 21
4.2 Initialization 25
4.3 Dialing 31
4.4 Terminal Mode 34
  4.4.1 The Main Menu 36
4.5 File Transfer Mode 38
  4.5.1 Apple => Host Transfer 38
  4.5.2 Host => Apple Transfer 42

5. Future Enhancements to TRANSPORTER 48

5.1 Improvements to Existing Features 48
5.2 Extensions 51
  5.2.1 A One Bit Sliding Window Protocol 51
  5.2.2 Listing Apple Directories 55

References 58

I. User Manual: 60

II. Appendix A 74

III. Appendix B 75

IV. Appendix C 76

Vita 79

iv
ABSTRACT

TRANSPORTER is a modem communication program for the Apple II microcomputer running under the Pascal operating system. It is designed to work with any modem connected to the Apple through the Apple Communication Serial Interface Card or with the Hayes Micromodem II.

TRANSPORTER makes it possible to use the Apple as if it were a video terminal connected to a remote computer system. It also supports bidirectional ASCII text file transfers.

TRANSPORTER was designed to be user friendly, to be as portable (to other microcomputers) as possible, and to maximize the ease of program development.

This thesis will address the following topics:

- the general structure of the program
- the data structures used
- descriptions of each phase of TRANSPORTER
- some possible enhancements to the program
1. Introduction

TRANSPORTER was originally developed so I could use an Apple II computer (Pascal Operating System version 1.1) as a video terminal. I had access to a Hayes Micromodem II, but no Pascal software to control it. The Micromodem's onboard ROM contains programs for setting the baud rate, originate or answer mode, and automatic answering and dialing, to name a few. The problem was that the Micromodem's firmware refers to locations and subroutines that only exist in the Apple DOS 3.3 operating system, not in the Pascal operating system [11].

After solving these problems I added procedures to support bidirectional file transfer between the Apple and remote (host) computers. This allowed me to edit files with the Apple's screen editor, transfer them to my directory on Lehigh University's DEC-20 computer, and have them printed out on the high speed line printer. As a matter of fact, this manuscript was prepared by using my Apple II and TRANSPORTER to enter and modify text on the DEC-20. Also, by using the DEC-20's mail program, I could "mail" a file to the directory of another Apple user. He could then transfer the file to his Apple. It became a very convenient way to trade software.
2. General Program Structure

The source code for TRANSPORTER is about 80K bytes. Since the largest file that Pascal’s editor can handle is approximately 18K bytes, it became necessary to divide TRANSPORTER into separate files. I wanted to do this in a way that facilitated program development and management. Two observations were apparent: TRANSPORTER has a modular structure and it contains procedures and functions that are useful to other programs as well. The modular structure suggested a logical division of the program into six files. Each of these files represent a distinct phase in the operation of TRANSPORTER. The procedures that could be utilized by other programs were inserted in the system library [3]. The system library contains code for procedures and functions. This code is available to any host program that uses the library, just as if the procedures were defined in the program itself.

This chapter will discuss the library units used, and the files of which TRANSPORTER is comprised.
2.1 Library Units

Apple Pascal allows a programmer to create Intrinsic Units [3] that can be added to the system library. A unit consists of four parts:

1. A heading

2. An Interface Part

   - Defines the procedures, functions and data that are accessible to the host program and to the unit.

3. An Implementation Part

   - Defines procedures, functions and data that are accessible only to the procedures (and functions) of the unit.

4. An Initialization Part

   - The "main program" of the unit. It consists of a BEGIN and an END with any number of statements between them, including the empty statement. This part of the unit is automatically executed at the beginning of the host program's execution.

The significant difference between a unit and an intrinsic unit is that an intrinsic unit's code is not inserted into the codefile of the host program. It is automatically loaded into memory at run-time. The code of a regular unit must be linked into the codefile of the
host program before the program is executed.

TRANSPORTER uses two intrinsic units that I have added to the system library:

1. Intrinsic Unit SCREEN
2. Intrinsic Unit MEMORYSTUFF

SCREEN contains routines that are useful for interactive I/O and writing menus to the CRT screen (see implementation of SCREEN in Appendix A). For example:

1. Procedure clearscreend;
   - Clears the screen and homes the cursor.

2. Procedure bspaces(n: integer);
   - Writes n nondestructive backspaces.

3. Procedure clear(column, row: integer);
   - Clears from the given coordinate to the end-of-screen.

SCREEN also contains two functions important to the operation of the program. They are:

1. Function getch(options: charset): char;
   - Gets one character from the keyboard. If the character is in options, getch converts it to upper case and returns it
to the calling procedure. If it's not in options then getch rings the bell and repeats the process.

2. Function readreal: real:

- Apple Pascal has a problem with the procedure read(x) if x is of type real. It doesn't support line editing features. Readreal solves this problem.

The intrinsic unit MEMORYSTUFF contains the procedure poke and the function peek. They are written in 6502 assembly language and work like BASIC's poke and peek.

2.2 Logical Division of Files

In Apple Pascal it is possible to split a large program into separate files and INCLUDE them in the compilation. As mentioned earlier, TRANSPORTER has been divided into six files that reflect five different phases of the program. These files are:

1. INIT.TEXT

- Contains all procedures for the initialization of global variables and for the initialization of the modems.

2. BLDPARAMS1.TEXT & BLDPARAMS2.TEXT

- Contain the routines that build the parameter files. The parameter files are
used to store information about a user's Apple and the host computers that he calls.

3. DIAL.TEXT

- Contains procedures that support dialing for manual dial modems and autodial modems.

4. TRANSFER.TEXT

- The file transfer routines.

5. TERMINAL.TEXT

- Contains the main program. All of the global data types, variables and constants are in this file, along with low-level procedures that are called by procedures in the other files.

- The procedure terminal is also in this file. Terminal is where TRANSPORTER spends most of its time. It is the procedure that lets the Apple act like a video terminal. I will frequently refer to the actions of this procedure as terminal mode.

except for the type charset which is defined in SCREEN
3. Major Data Structures

The data structures in TRANSPORTER can be categorized into three major classes:

- those that store information about the Apple environment and the various host computer environments

- those that deal directly with absolute memory locations for controlling and exchanging data with the modems

- buffers for holding data received from a file transfer and for writing that data to a diskette

This chapter will discuss the first two classes. A description of the buffers associated with file transfers can be found in section 4.5.

3.1 Environment Records

TRANSPORTER uses two records to inform it about the Apple it is executing on and the host computer it is communicating with. The methods used to gather the information that these records will store is discussed in section 4.1.
3.1.1 Apple Environment

The declaration of the first record, which defines the Apple environment, is as follows:

```
TYPE
modemtype = (hayes, applecommcard,
               smart300, smart1200);
envrnmntype = RECORD
  model: (plus, e);
  modem: modemtype;
  baudrate: (high, low);
  lowercasetranslation: boolean;
  numdrives: integer;
  defaultdrive: STRING;
END;

VAR
  apple: envrnmntype;
```

A distinction is made between the Apple //+ and the Apple //e only for the reason that the Apple //e has a full ASCII keyboard. Certain characters not available on the keyboard of the Apple //+ are generated by TRANSPORTER when the user types special control keys. These characters are listed in the help menu. Since the Apple //e generates all of the ASCII characters, the user does not need to see this portion of the help menu. TRANSPORTER decides how much of the menu to display by examining the value of model.

The modem field of the record is of type modemtype. Each value of modem does not necessarily represent a different modem. At this stage in the development of
TRANSPORTER the values are defined as follows:

- **hayes** represents the Hayes Micromodem II, an automatic modem (eg. automatic dialing) [8].

- **applecommcard** represents all of the manual modems (eg. the Novation D-CAT) that are connected to the Apple through the Apple Communications Interface Card [1].

- **smart300** represents the Hayes Smartmodem 300 [9]. This modem uses the Apple Communications Card but is automatic, thus requiring special attention by TRANSPORTER.

- **smart1200** represents the Hayes Smartmodem 1200, a 1200 baud automatic modem. I haven’t had access to one of these and so am operating under the assumption that it’s the same as smart300, only faster. TRANSPORTER is yet to be tested with this modem.

A user has the choice of transmitting at a high or low baudrate. If a low baudrate is chosen the Apple Communications Card (or the Hayes Micromodem II) is initialized to operate at 110 baud. If high is chosen the Communication Card (or Micromodem) is initialized to 300 baud.

The lowercase translation field of the record is completely ignored if TRANSPORTER is executing on an Apple //e. Its function is to determine if the Apple II+ should behave like a "normal" keyboard with SHIFT and CAPSLOCK keys. If its value is true then all alphabetic characters will be converted to lower case by TRANSPORTER before being sent to the host computer. A user can
capitalize a letter by typing Control-A. The next letter typed will be capitalized. CAPSLOCK is emulated by typing two Control-A's in a row. All letters will be sent to the host as upper case until another Control-A is typed. The reading and converting of keyboard characters is done by the function readkey which is written in 6502 assembly language to speed-up the process. Even so, there is so much checking that has to be done that a fast typist can out-type TRANSPORTER. To solve this problem you can trade in your "+" for an "e". If this isn't feasible it helps to set the value of lowercase translate to false. This eliminates the need for TRANSPORTER to emulate a real keyboard hence improving typing response.

The number of disk drives connected to the Apple is stored in the numdrives field. The volume number [5] of the default disk drive is stored in defaultdrive. The default drive is the one that TRANSPORTER uses when transferring files.

---

2 This can be done from the main menu.

3 Actually, it still has to check for the special control characters mentioned earlier.

4 More on this in section 4.5
3.1.2 Host Computer Environment

The record that stores information about the host computer environment is slightly more complicated than the one describing the Apple environment. It is defined as follows:

```
TYPE
  hostype = RECORD
    name,      
    logout,  
    phononenumber: STRING;
    duplex: (full, half);
    deletech: char;
    CRLF: boolean;
    abortch: char;
    eofmrk: RECORD
      cntrlch: boolean;
      eofch: char;
      eofstrng: STRING;
    END;
  END;

VAR
  host: hostype;
```

The first three fields of `hostype` are STRING types that contain the name of the host, its logout command, and its phononenumber. The Hayes Micromodem II uses the value of phononenumber to automatically dial the host. Manual dial modems do not use this field, but it is still a convenient way for a user to save the phonenumbers of the hosts that he calls.

The duplex field lets TRANSPORTER know if it should write characters to the screen as they are typed (half
duplex) or wait for them to be echoed back from the host (full duplex) before printing them.

A host computer either uses a backspace (ASCII 8) or a rubout (ASCII 127) to delete the last character received in its input buffer. This delete character is stored in deletech. The Apple II+ can only generate a backspace. When TRANSPORTER detects that a backspace has been typed, it converts it to the value of deletech. Of course, if deletech is a backspace no conversion takes place.

Some computers require that every time they receive a carriage return (ASCII 13) it must be followed by a linefeed (ASCII 10). If the value of CRLF is true, TRANSPORTER will send the host the extra linefeed every time the user hits the RETURN key. Otherwise, just a carriage return is sent.

The last two fields of hostype are only used for file transfers. On a transfer from the host to the Apple the host file is transmitted to the Apple by a "DRIVER program" that is executing on the host. The user can abort the transfer by typing Control-C. When this happens TRANSPORTER must tell the host to abort the execution of DRIVER. The value of abortch is the character that the host recognizes as the signal to abort a running process. For example, on the Hewlett Packard
HP-3000 it is a "break signal" and on the DEC-20 it is a Control-C. On a transfer in the other direction, TRANSPORTER must send an end-of-file marker to the host when it has finished sending the file. This tells the host's operating system to close the newly received file. Some host computers use a control character (eg. the DEC-20 uses Control-Z) to mark the end-of-file, while others use a character string (eg. the CDC CYBER-720 uses %EOF). The eofmrk field of hostype contains this information.

3.2 Altering Absolute Memory- the Free Union Record

Two methods are used to alter or examine the contents of the Apple's memory. The first method is straightforward. It uses the procedure poke to write a new value to an address, and the procedure peek to examine the contents of an address. Poke and peek are written in Apple Pascal's 6502 assembly language.

The following example shows how peek and poke are used. The REPEAT loop is a simplified version of the "main loop" of TRANSPORTER.

----------

5 i.e. terminal mode
CONST
KEYBRD = -16384; { C000 }

VAR
data: integer;
chars: SET OF char;

REPEAT
IF character received in modem
THEN
BEGIN
   modemchar := chr(peek(data));
   IF modemchar IN chars
   THEN
      write(modemchar);
   END;
IF peek(KEYBRD) > 127
THEN
BEGIN
   keyboardchar := readkey;
   WHILE modem isn't ready to send data DO
   IF character received in modem
   THEN
      BEGIN
         modemchar := chr(peek(data));
         IF modemchar IN chars
         THEN
            write(modemchar);
      END;
   { Now send the char typed to the modem }
   poke(data, ord(keyboardchar));
END;
UNTIL carrier is lost OR user quits;

The variable data is assumed to be initialized to the address of the location where all data is sent and received through the modem. The variable chars is the set of all characters that are allowed to be printed on the Apple’s screen. It contains all of the printable
ASCII characters in addition to a few control characters such as carriage return, formfeed, bell, and backspace. The constant KEYBRD is the Apple's keyboard data address. When a key is pressed, bit 7 of KEYBRD is set. Thus, for the program to sense if a key has been pressed it only needs to see if the value of KEYBRD is greater than 127.

Poke and peek provide a useful way to either alter or examine a full byte of memory. However, it is frequently necessary to access a single bit of some memory location. This is especially true when dealing with I/O control and status registers. Pascal provides a mechanism to achieve this. It is a variant record (i.e. union type) with no tag field (i.e. a free union record). TRANSPORTER uses two different free union record types. One is for retrieving and storing a byte of memory, the other is for setting (or resetting) a particular bit of a memory location. When used in conjunction, these two data structures provide a means to alter or examine any bit of the contents of a given memory location. For speed considerations in a real-time program, it may be more efficient to do all "bit-fiddling" in assembly language instead of Pascal. Consider the following

-----------------

6 more on this in Chapter 5 - "Future Enhancements to TRANSPORTER"
declarations and procedures, using variant records to relax type checking:

TYPE
  byte = 0..255;
  word = PACKED ARRAY[0..1] OF byte;
  bits = PACKED ARRAY[0..7] OF boolean;

VAR
  memcpy: PACKED RECORD
    CASE boolean OF
      true: (bytehalf: byte);
      false: (bithalf: bits);
    END;
  memory: RECORD
    CASE boolean OF
      false: (address: integer);
      true: (contents: word);
    END;

PROCEDURE alterbit(addr, bitnum: integer;
  state: boolean);
BEGIN
  memory. address := addr;
  memcpy. bytehalf := memory. contents[0];
  memcpy. bithalf[bitnum] := state;
  memory. contents[0] := memcpy. bytehalf;
END;

FUNCTION examinebit(addr, bitnum: integer):
  boolean;
BEGIN
  memory. address := addr;
  memcpy. bytehalf := memory. contents[0];
  examinebit := memcpy. bithalf[bitnum];
END;

The variable memory is either an integer or a pointer to a data object of type word, depending on how it is accessed. Similarly, the variable memcpy is
either of type `byte` or of type `bits`. Procedure `alterbit` fetches the contents of an address, alters one of the bits, and writes it back to the address. In the first statement, the `address` field of `memory` is assigned the value of the parameter `addr`. In the next line `memcpy` is assigned the value of `memory.contents^[0]`. The value of `contents` is the last value assigned to `address`. Since `contents` is a pointer it is the address. It points to two bytes (i.e. the array of type `word`). The first byte is the one that we're interested in, so we use `contents^[0]` to get it.

For example, to get the contents of address FF we could either write

```pascal
memory.address := 255;
memcpy.bytehalf := memory.contents^[0];
```

or

```pascal
memory.address := 254;
memcpy.bytehalf := memory.contents^[1];
```

Now that we've obtained a copy of a byte of memory, we need to alter one of its bits. This is done in the third line of `alterbit`. On the previous line we assigned a value to `memcpy.bytehalf`. This value is converted, by Pascal, into a value of type `bits` since we're accessing the `bithalf` field of `memcpy`. This allows us
to change the bitnum bit to the value of state. The values of bitnum and state are, of course, supplied by the programmer.

The function computed by examinebit should now be obvious.
4. Modes of Operation

It was mentioned in Chapter 2 that TRANSPORTER operates in five different phases, or modes. The first three modes are:

1. Initialization
2. Creation and/or modification of parameter files
3. Dialing

Actually, the initialization mode is entered twice. The first time is when variables that store the values of ASCII control characters are initialized. The second time is after TRANSPORTER has determined the Apple and host computer environments by reading the parameter files. TRANSPORTER then makes the appropriate initializations based on the environments under which it will operate. The dialing mode is the third stage. If it is successful, TRANSPORTER enters terminal mode, where it stays until the user decides to "escape" to the main menu. A variety of options are available from the main menu; most notably, the file transfer mode.

This chapter will discuss the technical details of each mode. A more descriptive (from a user's viewpoint) explanation can be found in the USER MANUAL.
4.1 Creation and Modification of Parameter Files

After the initialization of the ASCII control character variables, TRANSPORTER calls the function parameterset which is contained in the file BLDPARAMS2.TEXT. First, parameterset determines the number of disk drives connected to the Apple. It does this by using the predefined Apple Pascal procedure UNITCLEAR [6]. UNITCLEAR cancels all I/O operations to the specified I/O device, and resets the hardware to its power-up state. If the I/O device is not present, the predefined Apple Pascal function IORESULT returns a nonzero value. In fact, IORESULT returns an integer result after each I/O operation. If the operation was successful, IORESULT returns zero. Otherwise, a nonzero result that can be used to determine what I/O error occurred is returned. For example, if you try to write to a file that is on a write-protected diskette, IORESULT will return the value 16.

After determining the number of disk drives, TRANSPORTER searches each diskette (starting with the one in the boot drive) for the file APPLPARAMS.DAT until it is found or the number of drives to search is exhausted. The search is performed by using the reset procedure together with the IORESULT function (i.e., IORESULT = 0 if file is found). APPLPARAMS.DAT contains one record of
type envrmnttype (see section 3.1.1). That is, it contains a record of the Apple environment. If the file isn't found, TRANSPORTER creates it on the diskette in the drive of the user's choice. Each field in the record of the file is explained to the user, who enters the appropriate values when prompted for them. After all of the values are assigned, they are displayed on the screen. If the user chooses, he can modify any value(s) before the record is saved in the file. If the file is found, TRANSPORTER displays its contents on the screen. If the user decides to alter the value of one or more record fields, TRANSPORTER will ask if the altered record should be written to the file. If the answer is "yes" the new record is saved and the program continues operation with the new values assigned to the global variable apple. Otherwise, the file is not altered, the record is assigned to apple, and TRANSPORTER continues.

Next, TRANSPORTER searches for the file HOSTPARAMS.DAT. This file contains a record of type hosttype (see section 3.1.2) for each host computer environment that the user has defined. The search is performed identically to the way it was for APPLPARAMS.DAT.
The following example is a parody of the actual code that creates and modifies HOSTPARAMS.DAT:

search for file
IF file isn't found
THEN
create it
ELSE
BEGIN
listhosts;
write('DEFINE A NEW HOST? ');
IF getch(['Y', 'N']) = 'Y'
THEN
appendhostfile;
END;
choosehost(recnum);
IF user hasn't quit
THEN
alterhost(recnum);

The procedure listhosts displays the name of each host computer that is defined. If the user wants to define a new host, the procedure appendhostfile is called. This procedure appends as many host environment records to the end of the file as the user wants, limited only by the amount of disk space available. Procedure choosehost uses the predefined Apple Pascal procedures seek and get to gain access to records in the file. It starts by getting the first record, printing its name on the screen, and presenting the user with options to:

- examine the host record
- choose this host as the one to call
- quit the program
If the user decides to quit, the function `parameterset` is assigned the value `false` and `TRANSPORTER` will terminate. If a host is chosen, then the `VAR` parameter `recnum` is assigned the position of the record in the file. If the host is not chosen, then `recnum` is incremented and `choosehost` continues with the next record in the file. This process continues until either a host is chosen or the user quits. If the user does not choose a host and end-of-file is reached, then `recnum` is set to zero and the process is repeated. If the procedure `alterhost` is reached, `recnum` will be the position of the chosen record in the file. This makes it possible for `alterhost` to do a quick `seek` to the record and display it on the screen. The user can now alter any of the host parameters. If an alteration is made, the new record can be saved in the file. This choice is of course left up to the user.

In summation, the boolean function `parameterset` searches for two files named `APPLPARAMS.DAT` and `HOSTPARAMS.DAT`. If they are not found they are created. If they are found the user can modify any of the data that they contain. The record contained in `APPLPARAMS.DAT` is assigned to the global variable `apple`, and the record that the user chooses from `HOSTPARAMS.DAT` is assigned to the global variable `host`. When choosing the host record, the user has the option of quitting the
program. In this case parameterset is assigned the value false. The only other way that the function can be assigned this value is for an I/O error, that can't be corrected by TRANSPORTER, to occur (e.g. I/O error #64- "bad data on diskette"). TRANSPORTER continues executing only if parameterset is assigned the value true.

4.2 Initialization

Initialization mode is entered twice. The first time is just the trivial case when variables are assigned the values of ASCII control characters. The second time is after the environment records have been assigned to the global variables apple and host. This is when the modems are initialized. Which modem is to be initialized is determined by the modem field of apple.

The simplest case is when the modem is connected to the Apple Communications Card. The procedure initapplecomncard (listed on the following page) performs the initialization of this device.
PROCEDURE initapplecommcard;
BEGIN
  rewrite(remout, 'REMOUT:');
cr1 := -16242 + 16 * SLOT;
status := cr1;
data := -16241 + 16 * SLOT;
poke(crl, 3);  { ** Tnt the ACIA chip ** }
poke(crl, FSW);
IF apple. baudrate = low
THEN poke(crl, 82)  { ** 110 Baud ** }
ELSE poke(crl, 17);  { ** 300 Baud ** }
END { INITAPPLECOMMCARD };

The variables status/crl and data are integers that represent four hardware registers on the communications card. Unlike regular memory locations that have a single read/write cell, these locations consist of a pair of cells that are related to each other. The first pair reads the status and writes the controls to the Motorola 6850 ACIA (Asynchronous Communication Interface Adapter) chip. The second pair, data, is where characters received and sent through the modem are read and written.

The ACIA must be initialized (by setting bits 0 & 1 of cr1) before it can be used [21]. FSW is the Flag Status Word. It is a global constant whose value is 1, and represents the format of the data transmitted through the modem.

The formatting is controlled by bits 2, 3, and 4 of cr1.
If those bits are all reset (i.e. have value 0), then characters will have the format of:

- 1 start bit
- 7 bit character length
- even parity
- 2 stop bits

The baud rate is set depending on the value of `apple. baudrate`. 110 baud is set if the value of `baudrate` is low, otherwise 300 baud is set.

The `rewrite` statement is for automatic dialing modems. `remout` is a global variable of type "FILE OF char". `REMOUT:`, which stands for "REMote OUTPUT", is an Apple Pascal volume name. The volume number associated with `REMOUT:` is "#8:", which is really SLOT 2. The communications card (and the Hayes Micromodem II circuit board) is plugged into this SLOT. If you are familiar with the APPLE DOS Operating System, `rewrite(remout, 'REMOUT:')` is similar to the command PR#2 except that output is only sent to the I/O device in SLOT 2 when a `write(remout, ch)` is used. An example of the use of `REMOUT:` can be found in section 4.3.
The next example shows the initialization of the Hayes Micromodem II. It is similar to the initialization of the Apple Communications Card, but the memory locations are different:

```
PROCEDURE hayesinit;
BEGIN
  cr2 := -16251 + 16 * SLOT;
  cr1 := -16250 + 16 * SLOT;
  status := cr1;
  data := -16249 + 16 * SLOT;
  cr2copy. bytehalf := INITFLAG;
  setcr2(INITFLAG, true);
  IF apple. baudrate = low THEN
    setcr2(BITRATESSELECT, false)
  ELSE
    setcr2(BITRATESSELECT, true); { 300 baud }
    poke(cr1, 3); { Initialize the ACIA }
    poke(cr1, FSW); { Set character format }
END;
```

The first difference is the absence of the `rewrite` statement. Even though REMOUT: is associated with the Micromodem I/O card (since the card is in SLOT 2) the Pascal BIOS (Basic I/O System) does not use the Micromodem's firmware as it does the Apple Communication Card's firmware. This means that another method must be used to dial the phone. If you look closely you will notice the presence of a new variable, `cr2`. This is the memory location of the modem control port. Setting and resetting bit 7 of this register has the effect of taking the phone OFFHOOK and ONHOOK, respectively. This allows
us to send pulse signals over the line which dial the phone (more on this in section 4.3). In addition to using bit 7 (which is represented by the global constant OFFHOOK having the value 7) of CR2, TRANSPORTER also uses four more bits to control the operation of the modem. They too are represented by global constants whose values are the bit numbers. They are:

1. BITRATESELECT = 0

   - If this bit is set the modem transmits at 300 baud, otherwise transmission is 110 baud.

2. TRANSMITENABLE = 1

   - The modem transmitter is turned on when this bit is set.

3. MODE = 2

   - Selects originate mode when set, answer mode when reset.

4. INITFLAG = 3

   - Keeps firmware from applying default values if this bit is set. Throughout the operation of the program this bit should always be set.

The status register has a different address than it does on the Apple Communication Card, but the bits report
the same conditions of the ACIA chip. Here are the constants that represent the bits of the status register that are used by this program:

1. RECEIVERREGISTERFULL = 0

   - If this bit is set then a character has been received in the modem.

2. TRANSMITTERREGISTEREMPTY = 1

   - The ACIA transmitter is ready to transmit a character if this bit is set.

3. NOCARRIERPRESENT = 2

   - A carrier tone is present when this bit is reset.

The procedure setcr2 is similar to the procedure alterbit in the example given in section 3.2. It is as follows:

PROCEDURE setcr2(bit: integer; state: boolean);
BEGIN
   cr2copy. bithalf[bit] := state;
   WITH memory DO
      BEGIN
         address := cr2;
         contents^.[0] := cr2copy. bytehalf;
      END;
END;
The pair of statements (in procedure hayesinit)

```c
    cr2copy. bytehalf := INITFLAG;
    setcr2(INITFLAG, true);
```

has the effect of setting bit 3 and resetting all other bits in cr2. That is, the modem is initialized so that the firmware does not apply default settings.

The baud rate is now controlled through the cr2 register instead of cr1, but the ACIA is initialized in the same way that it was for the communication card.

### 4.3 Dialing

There are three boolean functions that support dialing: `manualdial`, `dialhayesmart` and `autodial`. These functions are all contained in the file DIAL.TEXT.

The simplest is function `manualdial`. It prints the message "PLEASE DIAL THE COMPUTER", waits 10 seconds for the user to dial the phone and asks him to type "R" if a carrier is received or "A" to abort the program. If the user types "R", `manualdial` is assigned the value `true` and TRANSPORTER enters terminal mode. Otherwise `manualdial` is assigned the value `false` and the program terminates.

The function `dialhayesmart` is used for the Hayes Smartmodem and is slightly more complicated. It prints the message "Enter Dial Command: " and then executes a loop that writes each character that the user types to
the file remout, which is associated with the I/O device in SLOT 2 (see section 4.2). The firmware in the Smartmodem takes care of the actual dialing. The function returns the value true if a connection is established, otherwise it returns false.

The last function, autodial, is used for the Hayes Micromodem II. It is the most complex of the three because of the problem with Pascal's BIOS and the Micromodem's firmware (this was discussed in section 4.2). The following is a simplified section of the main body of autodial:

```
BEGIN
  setcr2(MODE, true); { originate mode }
  setcr2(OFFHOOK, false);
  dial;
  write('Waiting for Carrier');
  time := 30; { seconds }
  WHILE (getstatus(NOCARRIERNPRESENT))
    AND (time > 0) DO
    BEGIN
      y := peek(data);
      wait(1.0);
      time := time - 1;
    END;
  IF getstatus(NOCARRIERNPRESENT)
  THEN BEGIN
    writeln('No Carrier Detected');
    setcr2(OFFHOOK, false);
    autodial := false;
  END
  ELSE BEGIN
    writeln('Connection Established');
    setcr2(OFFHOOK, true);
    setcr2(TRNSMITENABLE, true);
    autodial := true;
  END;
END;
```
The boolean function \texttt{getstatus} is similar to the function \texttt{examinebit} in the example in section 3.2. It returns the logical value of the specified bit in the ACIA status register (e.g., \texttt{NOCARRIERPRESENT} represents bit 2).

The procedure \texttt{dial} deserves particular attention. It takes the phone OFFHOOK, sets the modem to originate mode, and waits two seconds for a dial tone. It then executes the following loop (again, this is simplified somewhat):

\begin{verbatim}
FOR i := 1 to LENGTH(host. phonenumber) DO
BEGIN
   digit := host. phonenumber[i];
   dialdigit(digit);
END;
\end{verbatim}

As you can see, \texttt{dial} gets the digits of the phone number from the host environment record and passes them to the procedure \texttt{dialdigit}, which simulates rotary pulses needed for the actual dialing. It is presented on the next page in its entirety.
PROCEDURE dialdigit(digit: char);
VAR i, x: integer;
BEGIN
  IF digit IN ['0'..'9'] THEN
    BEGIN
      IF digit = '0'
      THEN x := 10
      ELSE x := ord(digit) - ord('0');
      FOR i := 1 TO x DO
        BEGIN
          setcr2(OFFHOOK, false); wait(0.05);
          setcr2(OFFHOOK, true);  wait(0.05);
        END;
    END;
  END;
END;

First, digit is converted into an integer and then the phone's receiver is toggled off and on hook in the REPEAT loop. This simulates rotary pulse dialing. It also works on push-button phones. With practice, you may be able to dial in this manner on your phone.

4.4 Terminal Mode

We've already seen a parody of terminal mode in the first example of section 3.2. From that example it appears that all that happens when TRANSPORTER is in terminal mode is that the modem is polled for data received from the host (and printed on the screen if there is any) and that the keyboard is polled for activity (and sent to the host if something was typed). Actually, because the Apple's keyboard is limited and since we have to be on the lookout for the command that
escapes to the main menu, there is more than meets the eye. The example on the following page shows the actual code used for terminal mode. Note the introduction of the procedures handlemenu, sendbreak, and sendchar.

```
writeln('Entering terminal mode');
writeln('Control-E Escapes to Main Menu');
REPEAT
  IF getstatus(RECEIVERREGISTERFULL)
  THEN
    BEGIN
      modemchr := chr(peek(data));
      IF modemchr IN chars
      THEN
        write(modemchr);
      END;
    IF peek(KEYBRD) > 177
    THEN
      BEGIN
        kybrdchr := chr(readkey);
        IF kybrdchr = CTRL
        THEN
          BEGIN
            kybrdchr := null;
            handlemenu;
          END;
        IF kybrdchr = break
        THEN
          BEGIN
            kybrdchr := null;
            sendbreak;
          END;
        IF kybrdchr = backspace
        THEN
          kybrdchr := host, deletech;
          sendchar(kybrdchr, true);
        IF (kybrdchr = CR) AND (host. CRLF)
        THEN
          sendchar(LF, true);
        END;
      END;
  UNTIL getstatus(NOCARRIERPRESENT);
writeln(bell, "***** Lost Carrier *****");
```

If you study the example given in section 3.2 you will notice that the code that polls the modem is...
virtually identical to the above example. The significant differences occur after the keyboard character has been read by the assembly language function readkey.

If the kybrdchr is a Control-E, TRANSPORTER escapes to the main menu. If a Control-W (ASCII 23) was typed then a break signal will be sent to the host by the procedure sendbreak. A break signal (a steady space tone transmitted approximately 0.2 seconds) is sent automatically by setting bits 5 & 6 of the ACIA control register, cri. Since sending characters to the modem is also done in file transfer mode, the WHILE loop and poke statement in the old example have been replaced by the procedure sendchar.

4.4.1 The Main Menu

The procedure handlemenu prints the main menu and responds accordingly to the user's selection. The selections are:

1) Transfer a file (no error correction)
2) Transfer a file (with error correction)
3) CHANGE Host Parameters
4) CHANGE Apple Parameters
5) Return to terminal mode
6) AUTOLOGOUT
7) List Apple Directory
8) ****** HELP ******

Selections 2 and 7 haven't been implemented as of this
writing. They will be discussed in Chapter 5. Selection 1 is the topic of the next section in this chapter.

Since the host and Apple environments have been explained at some length, it need only be said that by selecting 3 or 4 it is possible for the user to alter the values of (most of) the fields of the environment records.

If the user decides to return to terminal mode then handlemenu simply sends the host a carriage return and returns to where it was called from—terminal mode. The carriage return is sent so that the host’s operating system will echo back its prompt character, which lets the user know that he is back in terminal mode.

Selection 6, AUTOLOGOUT, sends the host the logout command that is stored in the host environment record. It then waits until the carrier is lost. The program is then terminated via the Apple Pascal procedure EXIT (the only one in the program).

The HELP selection prints characters that are not available on the Apple ][+ keyboard together with the keys to type in order to generate those characters. It then prints the following message:

Control-C aborts file transfers.
Control-W sends a break signal.
Control-E escapes terminal mode.

If an Apple //e is being used then only the last message
is displayed. After the user has perused the help menu he presses any key and is returned to the main menu.

4.5 File Transfer Mode

When file transfer is selected from the main menu, the user is asked the direction of the transfer. If Apple => Host is chosen then the procedure apple2host is called. Otherwise the procedure host2apple is called. Of the two procedures, apple2host is the simpler. I shall describe it first.

4.5.1 Apple => Host Transfer

Basically there are four steps that must be taken in order to transfer a file from an Apple Pascal diskette to the host computer. They are:

1. Determine which file is to be transferred, and locate it.

2. Start the host's "file copy" program. This program will copy any input received from the Apple into a file that it creates on disk (eg. this file is called COPY on the DEC-20).

3. Send the file.

4. Stop execution of the host's "copy" program by sending the end-of-file marker.

To determine which file is to be transferred, TRANSPORTER prompts the user for the name of the "Apple source file". In the Pascal operating system the syntax of file
specifications is

<volume number><filename><extension>

For example,

#5:TRANSFER.TEXT

specifies the TEXT file named TRANSFER on volume number 5 (i.e. Drive 2). Here is where the defaultdrive field of the apple environment record is used. When prompted for the name of the file the user need only type the filename. TRANSPORTER will concatenate the volume number and TEXT extension to it. The defaultdrive can be overridden if the user supplies a volume number (or diskette name). However, the TEXT extension is mandatory. If the user supplies some other extension (eg. CODE) the TEXT extension will still be concatenated. This ensures that only text files will be transferred.

It should be noted that in the Pascal operating system, all text files must have the TEXT extension.

Once the user enters the file name it is stored in the variable source, which is a STRING type. If the user just hits a carriage return (i.e. a null file name), source will have a length of zero. TRANSPORTER interprets this as a sign that the user didn’t really want to transfer a file. For instance, maybe he selected
the wrong option from the main menu. In this case the program returns to terminal mode. If, however, the length of source is greater than zero, TRANSPORTER tries to locate it by executing a reset(sourcefil, source), where sourcefil is a variable of type text. The reset is followed by checking the value of the IORESULT function which was described in section 4.1. If an I/O error has occurred (eg. file not in directory) then the appropriate error message is printed and the user is asked to retype the file name. This process is repeated until the file is located or until the length of source is zero.

Now that the Apple source file has been opened (via reset), the user must execute the host’s copy program. For the sake of a concrete example, assume that the host is the DEC-20, whose copy program is named COPY. TRANSPORTER will print the following message:

INITIATE THE DEC-20’s COPY PROGRAM:

Enter the full command used to copy input from the terminal to a disk file.

*** TYPE CONTROL-C TO ABORT TRANSFER ***
*** TYPE CONTROL-E WHEN HOST IS READY ***

TRANSPORTER now enters a "pseudo-terminal mode", if you will. It sends all keyboard data to the DEC-20 and prints all data received from the DEC-20. If a Control-C is typed the transfer is aborted, the source file is
closed, and the program returns to terminal mode. The proper response to the above message is

```
copy ttv: filename
```

The DEC-20 responds with

```
TTY: => FTT.FMAME
```

and is ready to accept input from the terminal (TTY:) to be copied to the file named FTT.ENAME, which it creates. The user now types Control-F and TRANSPORTER starts sending the Apple source file.

A line-by-line protocol is used in transmitting a file. One line of the file is sent followed by a carriage return (CR). TRANSPORTER then waits for the host to send a linefeed (LF) in response to the CR it received from the Apple. This means that the host has processed the line of data in its input buffer and is ready for more. There is also a timeout interval involved with the end-of-line handshake. If the LF hasn’t been received in approximately five seconds then TRANSPORTER assumes that

```
7
'''

and the length of this timeout interval may be too short if the host is experiencing heavy use. Please see Chapter 5 - "Future Enhancements to TRANSPORTER" for details.
it was either lost or garbled during transmission. In this case the next line of the file is transmitted. This process is repeated until the end-of-file is reached or until the user aborts the transfer (by typing Control-C).

It is now time to inform the host that the transfer has ended so that it can close the file, suspend execution of the copy program, and return to the command level of its operating system. TRANSPORTER does this by sending the host's end-of-file marker, which is found in the host environment record. Finally, the Apple file is closed and TRANSPORTER returns to terminal mode.

Appendix B contains a listing of the code that implements the actual file transfer.

4.5.2 Host => Apple Transfer

The protocol for receiving a file from the host is considerably more complex than the protocol for sending a file. Before discussing it I would like to list the steps that must be taken in order to achieve a successful transfer in this direction:

1. Open a file for writing. The name of the file is chosen by the user.

2. Allow the user to start the DRIVER program executing on the host. This program reads the file specified by the user and sends it to the Apple, maintaining synchronization at all times.
3. Provide an orderly exit from the transfer in the event of a disk I/O error.

The technique used for opening the new file is similar to the one used for an Apple => Host transfer. Again, the file is opened on the disk drive specified by the defaultdrive field of the host environment record, and can be overridden by the user. The user need only type the file name in response to the prompt:

APPLE
DESTINATION FILE =>

If the TEXT extension isn’t supplied, it will be concatenated to the end of the file name. Also, as in the Apple => Host case, the transfer is aborted if a null file name is given. Once the file name is entered, TRANSPORTER opens it via the reset procedure. The status of the reset is then checked by the IORESULT function. If no error is detected by IORESULT it means that the file name already exists in the directory. The user is forced to reenter the file name until the name given is not found in the directory. Only then will TRANSPORTER open the file for writing via the rewrite procedure.

The transfer is initiated when the user starts the execution of the DRIVER program, which resides on the host. The protocol used for initiating the DRIVER
program is as follows:

1. TRANSPORTER enters a "pseudo-terminal mode" and the user enters the appropriate commands for executing DRIVER.

2. DRIVER displays the prompt:

   TYPE S TO START THE TRANSFER =>

If anything but an "S" (or "s") is typed, DRIVER will display the message:

   NO TRANSFER STARTED...TYPE CONTROL-C

and then terminate, returning the host to the command level of its operating system. If an "S" (or "s") is typed, then DRIVER sends TRANSPORTER startchar (ASCII 21), waits about 1 second, and then starts transmitting the file. NAK (ASCII 21) was chosen for startchar because it is one of the few ASCII control characters that the DEC-20’s operating system doesn’t transliterate into two printable characters (e.g., ASCII 3 is represented by the two characters "\xC3\x94\xC2\x82"). Meanwhile, TRANSPORTER is waiting to receive startchar from the host. On reception of this character, TRANSPORTER readies itself to receive the file. If startchar is not detected, TRANSPORTER will stay in "pseudo-terminal mode" and simply display the file it is receiving from DRIVER on the screen. For reasons that will become apparent after the operation of DRIVER is explained, a user will be able to detect that TRANSPORTER is stuck in pseudo-terminal mode. I opted not to incorporate a timeout interval while waiting for the host to send startchar expected by TRANSPORTER. The time interval in question is bounded by a probabilistic limit since it is dependent upon the response time of the user (to actually enter the command that will start the execution of DRIVER) plus the time that it takes the host to
compile and execute DRIVER.

DRIVER is now sending the file, assuming that the user typed "S" to start the transfer. Whether or not TRANSPORTER is stuck in pseudo-terminal mode or has started executing the procedure receivefile is not important at the moment. DRIVER simply transmits the first APPLEBUFFSIZE characters in the file and then waits for TRANSPORTER to send wakeupchar. APPLEBUFFSIZE is a constant with a value of 1024 (1K bytes) that must be the same value as its counterpart BUFFSIZE in the procedure receivefile of TRANSPORTER. TRANSPORTER sends wakeupchar (which was arbitrarily chosen to be an ASCII 33) after it has flushed its input buffer to the new file, signifying that it is ready to start receiving the next block of the file. If wakeupchar is lost or garbled in transmission then TRANSPORTER will time-out waiting for DRIVER to start sending. In this case, TRANSPORTER retransmits wakeupchar.

If TRANSPORTER is stuck in pseudo-terminal mode it will not send wakeupchar. Since DRIVER won't start sending the next block until it receives wakeupchar, the transfer will be stalled. When the user notices that the next block of the file is not being transmitted (since nothing is being displayed on the screen) he can type
Control-C to abort the transfer. I should mention that this has never happened in the five months that I have been using the program. But, even if TRANSPORTER should get stuck in pseudo-terminal mode, not much time is wasted since the problem is revealed in the time it takes to transmit the first block (1024 bytes) of the file (about 35 seconds at 300 baud).

While DRIVER is sending a block of the file, TRANSPORTER is storing it in a buffer. When the buffer is filled (1024 bytes, of course) TRANSPORTER compresses the data it contains and stores the result in the buffer named filebuff. The data is compressed by replacing all leading blanks of a line by an ASCII (16) followed by an ASCII (# of blanks). This conforms to the format of an Apple Pascal text file [7]. Next, filebuff is written to diskette. If an I/O error (such as "Insufficient space on diskette") occurs the transfer is aborted (The transfer can also be aborted at any time by the user). In the event of an abortion, TRANSPORTER stops the execution of DRIVER by sending the character stored in the abortch field of the host environment record. Then, before returning to terminal mode, TRANSPORTER gives the user the option of saving the contents of its input buffer in addition to what has already been written to the file.
If no I/O error occurred when TRANSPORTER wrote the contents of \texttt{filebuff} to the file, then TRANSPORTER sends \texttt{wakeupchar} to DRIVER and gets ready to receive the next block. If for some reason DRIVER doesn’t detect \texttt{wakeupchar}, TRANSPORTER will timeout and retransmit it.

DRIVER keeps sending the file a block at a time until the end-of-file is reached. DRIVER then sends \texttt{eofch} (which is a NAK for the same reason that \texttt{startchar} is) and terminates. When TRANSPORTER detects \texttt{eofch} it flushes its buffer to the file, closes the file, and returns to terminal mode. If TRANSPORTER does not detect \texttt{eofch} then, since DRIVER has terminated and the host has returned to the command level of its operating system, the user will eventually notice that something is amiss. In this case the user simply aborts the transfer. But the protocol hasn’t really failed since TRANSPORTER will give the user the option of saving everything that has been transferred so far, which happens to be the complete file.

A listing of DRIVER can be found in Appendix C. It is written in Pascal to facilitate portability. Because of the large size of procedure \texttt{host2apple}, I have decided not to provide a listing of it in this paper.
5. Future Enhancements to TRANSPORTER

This chapter is dedicated to outlining methods of improving existing features of TRANSPORTER and to suggesting guidelines for future enhancements to the program.

5.1 Improvements to Existing Features

When TRANSPORTER detects that a Control-W has been typed it sends the host a break signal. The decision to "hardwire" Control-W as the BREAK key was inflexible and may deprive the user of some special features provided by the host. For example, the DEC-20 recognizes a Control-W as the command to delete the last word typed in its input buffer. Flexibility can easily be gained by adding a new field to the host environment record and letting the user decide its value. The new field might be called breakchar. Of course, the value of breakchar should not conflict with the control characters used to generate the characters that aren't available on the Apple keyboard.

Another candidate for inclusion in the host record is a timeout field. This field would specify the length of the time-out interval used in the end-of-line handshake during Apple => Host file transfers. An appropriate value for timeout could possibly be assigned by asking the user if the host is experiencing "heavy",
"moderate", or "light" system use. The best time to ask this question is probably at the beginning of each Apple => Host file transfer since system use is not constant.

One last improvement would be to write terminal mode as an assembly language procedure to improve keyboard response. This may be very hard to do since terminal mode calls on the procedure handlemenu. The problem is that the Apple Pascal 6502 assembler doesn’t provide a mechanism for letting a .PROC or .FUNC (what the assembler calls procedures and functions) call a Pascal procedure. A possible solution comes to mind (assume terminal mode is named .PROC term):

Have TRANSPORTER pass handlemenu to term as a parameter. That is, term receives the address of handlemenu.

The problem with this solution is that it isn’t. Apple Pascal doesn’t allow procedures to be passed as parameters. Perhaps the best solution would be to write two assembly language .FUNC’s to replace the calls on getstatus and peek in terminal mode (see listing in section 4.4). They would be defined as follows:

1. .FUNC modemready

- returns the value 1 (true) if a character
has been received in the modem (i.e. bit 0 of status is set), otherwise returns 0 (false)

2. .FUNC keyboardready

   - returns 1 if bit 7 of KEYBRD is set, otherwise returns 0.

Modemready will execute faster than function getstatus because getstatus is a Pascal function that compiles into P-code that must be interpreted. Also, getstatus must compute which bit of status it is to evaluate; modemready returns the value of the same bit every time. For similar reasons, keyboardready will execute faster than the function peek. Peek, although written in assembly code, must store 2 bytes (the parameter that is the address to "peek") in 2 consecutive temporary memory locations and access memory indirectly by using the contents of the temporaries as a pointer. In addition, the P-code interpreter must evaluate the code generated by the expression

   IF peek(KEYBRD) > 127

On the other hand, keyboardready can be thought of as "hardwired". That is, no parameters are passed to it since it will always retrieve the contents of KEYBRD. Not only this but since keyboardready returns a boolean
value, the P-code interpreter's job is simplified since it only has to evaluate the code generated from

IF keyboardready

5.2 Extensions

In section 4.4.1 I stated that two options from the main menu are yet to be implemented. To refresh your memory, they are:

1. File transfers with error recovery
2. Providing the user with the ability to see Apple directory listings.

In this section I would like to suggest a protocol for file transfers with error recovery, and suggest a method for obtaining a directory listing.

5.2.1 A One Bit Sliding Window Protocol

When transferring a file, the sender usually divides the file into blocks and transmits them individually to the receiver. When error recovery is added to the transfer protocol, control information must be supplied with the data block. The sender would then transmit a frame consisting of the control information and the data block. The control information must at least include a checksum for detecting transmission errors and a sequence number so that the sender and receiver can keep track of
which data block is being transmitted. It should also contain an acknowledgement so the receipt of an undamaged frame can be acknowledged. The frame might be defined as follows:

```
frame = RECORD
    segno: 0..n;
    ack: 0..n;
    data: PACKED ARRAY[1..SIZE] OF char;
    chksum: integer;
END;
```

In a sliding window protocol [10] the sender maintains a list of consecutive sequence numbers that correspond to frames it is permitted to send, and the receiver maintains a list of consecutive sequence numbers that correspond to the frames it is permitted to receive. These lists are called the sending window and receiving window. The sender is permitted to send any frame whose sequence number falls within the receiving window. As frames are transmitted the sequence numbers are incremented (modulo n + 1). Hence the term sliding window.

A one bit sliding window (n = 1) uses one bit for the sequence number. Thus a frame is numbered 0 or 1.
The protocol, assuming no transmission errors, works as follows:

At the start of the transfer the receiving and sending windows both contain 0. Frame 0 is sent. The receiver gets it and slides its window up to 1. The receiver sends ack = 0 to the sender. The sender receives the ack and knows that it is for frame 0 because ack = 0. The sender slides its window up to 1 and sends frame 1. The receiver gets it, slides its window up to 0 (remember, modulo 2 addition), and sends ack = 1. The sender receives the ack, slides its window up to 0, and sends frame 0 (the third frame sent so far), etc.

This protocol is of the stop-and-wait genre. The sender transmits a frame and waits to receive an acknowledgement before sending the next frame.

If a frame arrives with an error in the data block, the receiver doesn’t send the ack and doesn’t slide its window. The sender eventually times-out waiting for the ack. But since the sending window isn’t changed until an ack is received, the sender will retransmit the last frame. When the receiver gets the retransmitted frame it knows that it hasn’t saved it (on disk) since the sequence number falls within the receiving window.
Consider what happens when an acknowledgement is lost in transmission:

Suppose the receiving window contains 0 and the sender has just sent a frame with sequence number 0. The receiver gets it, finds that it is error free, saves it, slides its window up to 1, and sends ack = 0. A noise burst hits the line and the ack is wiped-out. The sender times-out and retransmits frame 0. The receiver gets it, realizes that the sequence number falls outside its window, and so discards it. An ack = 0 still must be sent though, or the sender will repeatedly time-out and retransmit the same frame. Eventually the sender will receive ack = 0, slide its window up, and send the frame expected by the receiver.

So, we have seen that damaged frames and lost acknowledgements do not cause the protocol to fail.

Suppose the sender’s time-out interval is a little too short, and that the receiver has just sent ack = 0 for frame 0. The sender times-out while the ack is still on the line and retransmits frame 0. The receiving window contains 1, so the receiver rejects frame 0 but still sends ack = 0. Meanwhile the original ack = 0 arrives at the sender so the sender slides its window and sends frame 1. Next, the sender gets the second ack = 0. Since the sending window holds a 1 the sender will not confuse ack = 0 for frame 1. That is, the sender will not slide its window and send the next frame.

In summary, the one bit sliding window protocol will
not fail because of errors in the data block, lost acknowledgements, or premature time-outs.

5.2.2 Listing Apple Directories

It would be helpful for a user to be able to see an Apple directory listing. This option should be available from the main menu and also immediately before the user is prompted for file names in file transfer mode. I would like to just briefly sketch an outline of a method for obtaining directory information from a diskette.

A directory starts on block 2 of a diskette, and can be thought of as consisting of two parts:

1. A header

- contains the diskette name, creation date, number of blocks used, number of files, and other information

2. File information

- contains file names, the block numbers where a file starts and ends, the file type (text, code, data, etc.), the date the file was last written, and other information

Assuming we know the exact format of the directory, we

---

8 I have tested this method successfully.
could define two records whose structures exactly match the structure of the directory. Suppose that the following declarations are given:

```
TYPE
  xdate = PACKED RECORD
    month: 0..15;
    day: 0..31;
    year: 0..127;
  END;

filetype = (xnull, bad, code, text, info, data, graf, foto);

vheader = PACKED RECORD
  endblock, startblock: integer;
  dummy1, dummy2: 0..255;
  vname: string[7];
  nfiles, nblocks: integer;
  dummy3: integer;
  creat: xdate;
  dummy4, dummy5: integer;
END;

direntry = RECORD
  endblock, startblock: integer;
  ftype: filetype;
  name: string[15];
  eod: integer;
  wlast: xdate
END;

dirarray = ARRAY[1..77] OF direntry;
```

VAR
  volinfo: vheader;
  directory: dirarray;
  dirmark: integer;
  dirfile: file;

The following example will put the directory of the diskette (in the drive specified by volnum) onto the heap.
by using the reset procedure. It will then set the
pointers volinfo and directory to the proper locations:

(*$I- *) { turn off compiler I/O checking }  
reset(dirfile, volnum);  
(*$I+ *)  
IF IORESULT = 0  
THEN  
BEGIN  
mark(dirmark);  
new(volinfo);  
new(directory);  
close(dirfile);  
END;

After displaying the directory on the screen, the space
on the heap should be deallocated by

release(dirmark)
References


   1978.
   op. cit. pg. 18.

   op. cit. pp. 75-81.

   1980.
   op. cit. pp. 63-64.


   1980.
   op. cit. pg. 43.

   1980.
   op. cit. pg. 13.


[10] Tanenbaum, Andrew S.
    Software Series: Computer Networks.
    Prentice Hall, 1981.
A Pascal Library Unit for the Micromodem II.
I. User Manual:

TRANSPORTER is a Pascal program that transforms an Apple I[+ or Apple //e (running the Pascal operating system) into a smart video terminal that can communicate with timesharing computer systems and computer "bulletin boards". If you have a modem connected to the Apple Communications Serial Interface Card (such as the Novation D-CAT or Hayes Smartmodem 300) or the Hayes Micromodem II, TRANSPORTER will provide you with the capability of taking advantage of the remote computer system's software base and peripheral devices (such as high-speed line printers). TRANSPORTER also supports text file transfer. This allows you to do all editing in the Pascal editor and transfer the finished product to the remote system, thus minimizing connect-time charges. File transfer capability also enables you to trade software with other Apple Pascal users who also use the remote computer system.

TRANSPORTER was designed to be easy to use. After your Apple has been connected to the remote computer (often called the host computer) you only have to remember that typing a "Control-E" will present you with a menu from which a variety of options are available.
STARTING-UP

TRANSPORTER is stored on a "turnkey" diskette, which means that all you have to do to run the program is insert the diskette in drive 1 and turn on the Apple. If this is the first time that you run the program you must insert a diskette in every other disk drive that you may have. It doesn't matter what diskettes you use. Any Pascal diskette will suffice. The reason is that TRANSPORTER will automatically count the number of disk drives that are connected to your Apple. In order to do this correctly, each drive must be "on-line". Once the number of drives is determined, TRANSPORTER will search each drive for the file named APPLPARAMS.DAT. This file contains information about the hardware configuration of your Apple. If it isn't found (which it won't be if this is the first time that you're running the program), TRANSPORTER will tell you that it is creating it and ask you the volume number of the disk drive which contains the diskette that you want to save it on. The most logical place to save APPLPARAMS.DAT is on the diskette with TRANSPORTER, which is in the boot drive.
For those of you that aren't familiar with volume numbers and the disk drives that they specify, they are as follows:

<table>
<thead>
<tr>
<th>Volume</th>
<th>Drive</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1 (boot drive)</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
</tr>
</tbody>
</table>

Now that you have told TRANSPORTER where it is to save APPLPARAMS.DAT, you will be led through a series of questions whose answers will be stored in the file. For the most part, the answers to the questions are obvious. Here are explanations for those that aren't:

**Question:** Convert all the letters that you type into lower case?

**Answer:** If you are using an Apple ][+, TRANSPORTER will read the keys that you type a little faster if you answer No. However, if you answer Yes, all of the alphabetic keys that you type will be converted to lower case before they are sent to the host computer. You will now be able to capitalize a letter by typing "Control-A" before you type the letter. You can also emulate a CAPSLOCK key by typing 2 Control-A's in a row. From then on, all of the alphabetic keys that you type will be sent to the host as upper case. To stop emulation of CAPSLOCK, simply type another Control-A.

**Question:** What would you like the default disk drive to be?

**Answer:** This question is only asked if you have
more than one disk drive. You should enter the volume number of the drive that you want TRANSPORTER to default to when reading or writing a file during a file transfer. I choose volume number 5 because it is where I prefer files to be transferred to or from. In any event it is only a default value. You will have full control when the time comes.

After you have answered all of the questions, APPLPARAMS.DAT will be written to the diskette in the drive that you specified and TRANSPORTER will display the information on the screen. Now is your chance to change any of the answers that you may have made mistakes on. If you do make any changes you will be asked if they should be saved in APPLPARAMS.DAT. Whether you decide to save these changes or not, TRANSPORTER will continue operation using the values you changed in its data base.

Now, TRANSPORTER searches for the file named HOSTPARAMS.DAT. If it isn't found you get to specify which diskette it is to be created on, and (as it was when creating APPLPARAMS.DAT) you will be led through a series of questions that will define each host computer that you use. Here are explanations of some of the less obvious questions that you will be asked:

**Question:** Does the host expect to receive a LINEFEED after each CARRIAGE RETURN that you type?

**Answer:** Most host computers echo a linefeed upon receiving a carriage return. If you
happen to be using a host that doesn't, answer Yes to this question.

**Question:** Does the host computer use a String or a Control character to mark End-of-File?

**Answer:** This information must be supplied in order to successfully transfer a file from the Apple to the host. Many computers (such as the DEC-20) use the control character "Control-Z" (whose ASCII value is 26). The CDC CYBER-720 uses the string "%EOF", while the Hewlett Packard HP-3000 uses the string ":". If you don't know the answer you should find out by any means possible to you.

**Question:** Does the host computer use a break signal or a control character to abort a running program? <R>reak  <C>ontrol  < ? >

**Answer:** If the host uses a control character, type "C". You will then be asked to enter the ASCII value of the character. If, however, the host requires a break signal, type "R". You will then be told that a break signal is generated when you type a "Control-W". TRANSPORTER uses this information to automatically abort the program that sends the Apple a file during a host to Apple file transfer (if and when you decide to abort a transfer in this direction).

After you've answered all of the questions, the information is saved in HOSTPARAMS.DAT. You can define as many host computers as you desire. Once APPLPARAMS.DAT and HOSTPARAMS.DAT have been defined, TRANSPORTER will retrieve the necessary information from them instead of asking you to supply the information each time that you run the program.

Now it is time for you to decide which host computer you want to call. TRANSPORTER will display the message
****** Choose a Host ******

at the top of the screen and list the name of the first host defined in HOSTPARAMS.DAT, followed by 4 options for you to choose from. For example, if the first host that you defined was the DEC-20, you will see the following:

****** Choose a Host ******

DEC-20  <Y>es <E>xamine  
        <N>o  <Q>uit    < ? >

If you type:

- "Q", TRANSPORTER will terminate with the message:

PROGRAM ABORTED
Put boot diskette in
Drive 1 and type R to
return to Pascal:

You then type R and are returned to the command level of the Pascal operating system.

- "E", TRANSPORTER will display the information you provided about the DEC-20 and ask if you want to "CALL THIS HOST?". If you answer No, TRANSPORTER will list the name of the next host you defined followed by the same options. If you only defined one host or if all of the hosts have already been listed, then TRANSPORTER starts listing from the beginning.

- "N", TRANSPORTER lists the name of the next host

- "Y", TRANSPORTER will display information about
the host and ask you if any of the information should be changed. If you decide to change something you have the option of storing the changes in HOSTPARAMS.DAT.

Now that you have chosen a host, it is time to dial the phone and connect to the host computer system.

**DIALING**

If you are using a manual-dial modem that is connected to the Apple through the Apple Communications Serial Interface Card, TRANSPORTER will ask you to "PLEASE DIAL THE COMPUTER:", wait 10 seconds, and display the message:

```
Type:  <R>eceived Carrier
       <A>abort
```

If you do not hear a carrier signal you should type "A" to terminate the execution of TRANSPORTER.

If you are using the Hayes Smartmodem 300, TRANSPORTER will ask you to "ENTER THE DIAL COMMAND:"
You respond by typing the appropriate Smartmodem dialing command, followed by a carriage return. Then respond accordingly to the message:

```
After dialing is finished,
Type:  <C>onnected
       <N>o connection made
```

If you are using the Hayes Micromodem II, just sit
back and relax; everything is automatic.

**TERMINAL MODE**

Terminal mode is the operational phase of TRANSPORTER where you treat the Apple as if it were a "dumb" video terminal. If dialing was successful and a connection was made with the host computer, the message

Connection Established...Entering terminal mode

Control-E escapes to the main menu

will be displayed. You now log on to the host in the normal fashion. To see the menu type a "Control-E". You will be presented with the following options:

**** Menu ****

1) File Transfer (no error recovery)
2) File Transfer (error recovery)
3) CHANGE Apple Parameters
4) CHANGE Host Parameters
5) Return to terminal mode
6) AUTO-LOGOUT
7) LIST Apple Directory
8) **** HELP ****

Options 2 and 7 have not been implemented yet. Options 3 and 4 allow you to change some of the values of the Apple and host parameters that you defined earlier when creating the files APPLPARAMS.DAT and HOSTPARAMS.DAT. Options 5 and 6 are self-explanatory. This leaves us with options 1 and 8. Choosing option 8 will cause the
help menu to be displayed. If you are using an Apple //e
the help menu will appear as:

Control-C aborts file transfers.
Control-E escapes to the main menu.
Control-W sends a break signal.

If you are using an Apple II+, the help menu, in addition
to displaying the above message, also displays a list of
characters that are not available on the Apple II+[+ key-
board together with the keys that must be typed to
generate them.

FILE TRANSFER

Choosing option 1 from the main menu enables you to
transfer text files between the Apple and host computers.
If you type the number "1" from the main menu TRANSPORTER
will display:

FROM APPLE TO HOST?

If you answer No, TRANSPORTER will ask:

FROM HOST TO APPLE?

If you answer No again, TRANSPORTER returns to terminal
mode.
If you decide to transfer a file from the Apple to the host, TRANSPORTER will display:

APPLE SOURCE FILE NAME
=>

There are two ways for you to enter the file name. You can use the long form or the short form. Suppose the name of the file that you want to transfer is MYFILE.TEXT and that it is on the diskette in drive 2. You would enter:

APPLE SOURCE FILE NAME
=> #5:MYFILE.TEXT

If the default drive volume number is 5, you would only have to enter:

APPLE SOURCE FILE NAME
=> MYFILE.TEXT

Actually, you don’t have to supply the "TEXT" extension because TRANSPORTER automatically adds it to the file name if you don’t (or if you supply any other extension such as "CODE"). If the file is not found you will be told so and asked to try again; until the file is found or until you enter a "null" file name (i.e. a carriage return). Once the file has been found, you will be told to initiate the execution of the host’s "copy" program.
This is a program that copies input from an external device (in this case the Apple) to the host's disk storage. If you are using the DEC-20 you would see the following:

INITIATE THE DEC-20's COPY PROGRAM:

Execute the program that copies input from the terminal to disk.

** TYPE CONTROL-C TO ABORT TRANSFER **
** TYPE CONTROL-E WHEN HOST IS READY **

You would now type:

COPY TTY: NEWHOSTFILE

This command will cause any input from the terminal (TTY:) to be copied to the file NEWHOSTFILE. The DEC-20 will respond with:

TTY: => NEWHOSTFILE

The DEC-20 is now ready to receive the file, so you type a "Control-E" and the transfer begins. The file is displayed on the screen as it is being transferred to the host. When the transfer is completed TRANSPORTER displays the message:

** END OF TRANSFER **

You are now back in terminal mode. You may abort the
transfer at any time by typing "Control-C". TRANSPORTER will abort the copy program by sending the host's end-of-file marker and then display:

*** TRANSFER ABORTED ***

before returning to terminal mode.

If you decide to transfer a file from the host to the Apple, TRANSPORTER will display:

APPLE DESTINATION FILE NAME =>

Again, you can take advantage of the default drive and automatic addition of the ".*EXT" extension and just type in the file name. If the file name you supply already exists in the Apple directory, TRANSPORTER will tell you so and ask you to reenter the file name; until you supply a unique name or a "null" name. If you enter a null file name you will be returned to terminal mode. Otherwise, TRANSPORTER will display:

EXECUTE THE DRIVER PROGRAM

** TYPE CONTROL-C TO ABORT TRANSFER **

The DRIVER program is one that comes with TRANSPORTER. You must have this program on your (host) directory, tailored for the particular host computer on which it
will execute. As of this writing a version exists only for the DEC-20. However, DRIVER is written in Pascal and should only require minor changes dealing with the way it handles I/O to get it to execute on the host computer that you use.

The first thing that DRIVER does, after you start it executing, is print the message:

```
TYPE S TO START TRANSFER =>
```

You enter the proper response and the transfer begins. The file will be displayed on the screen as it is received at the Apple. Do not get disturbed when the transfer pauses periodically. TRANSPORTER is just preparing the data it is receiving to be written to the diskette. Occasionally you'll hear the whir of the disk drive as TRANSPORTER writes a portion of the file. If at any time you want to abort the transfer, type "Control-C". TRANSPORTER will print the message:

```
** TRANSFER ABORTED **
```

Save what has been transferred so far?

If you answer No, whatever has been written to the diskette will be purged, otherwise it will be saved. The transfer will automatically abort if there is not enough
room on the diskette. Again, you will have the option of saving whatever has been written up to that point. If the transfer has been successfully completed, TRANSPORTER displays:

** TRANSFER COMPLETED **

and returns you to terminal mode.
II. Appendix A

EXAMPLE OF A LIBRARY UNIT

UNIT SCREEN;
INTRINSIC CODE 23;

INTERFACE

TYPE
  charset = SET OF char;

PROCEDURE skiplines(n: integer);
PROCEDURE home;
PROCEDURE reverseflf;
PROCEDURE cleartoend;
PROCEDURE clear(x, y: integer);
PROCEDURE clearscreen;
PROCEDURE toupper(VAR kh: char);
FUNCTION getch(chset: charset): char;
PROCEDURE spaces(n: integer);
PROCEDURE bspaces(n: integer);
FUNCTION readreal: real;

IMPLEMENTATION

{ The code for procedures & funcitons goes here }

BEGIN

{ Any initialization of variables belongs here }

END { UNIT SCREEN }.
III. Appendix B

APPLE TO HOST TRANSFER

BEGIN { SENDFILE }
   WHILE (NOT eof(sourcef1)) AND (NOT abort) DO
      BEGIN
         WHILE (NOT eoln(sourcef1))
            AND (NOT abort) DO
               BEGIN
                  read(sourcef1, ch);
                  sendchar(ch, false);
                  write(ch);
                  IF peek(KEYBRD) > 127 THEN
                     userabort(abort);
               END { WHILE >;
               IF NOT abort THEN
                  BEGIN
                     sendchar(CR, false);
                     IF host. CRLF THEN
                        sendchar(LF, false);
                        writeln;
                  { Time for some HANDSHAKING with the host }
                     time := 0;
                     REPEAT
                        echochar := chr(peek(data));
                        time := time + 1;
                        IF peek(KEYBRD) > 127 THEN
                           userabort(abort);
                     UNTIL (echochar = LF)
                     OR (time = TIMEOUT) OR (abort);
                     readln(sourcef1);
                     wait(0.1); { IN CASE HOST IS SLOW }
                  END;
                  END { WHILE };
                  wait(0.1);
                  sendeof;
               close(sourcef1);
      END { SENDFILE };
IV. Appendix C

IMPLEMENTATION OF DRIVER FOR THE DEC-20

PROGRAM driver(hostfile);

CONST
  PAUSELEN = 500000;  { about 1 sec. on DEC-20 }
  APPLEBUFFSIZE = 1024;

TYPE
  index = 0..APPLEBUFFSIZE;

VAR
  i: integer;
  j: index;
  ch, wakeupchar, eofchar: char;
  terminal: FILE OF 0..255;
  hostfile: text;

PROCEDURE wait;

VAR
  k: integer;
  applechar: char;

BEGIN
  REPEAT
    get(terminal);
    applechar := chr(terminal MOD 128);
    UNTIL applechar = wakeupchar;
    FOR k := 1 TO PAUSELEN DO ;
  END { WAIT };
PROCEDURE checkbuffsize(VAR j: index);

BEGIN
  IF j = APPLEBUFFSIZE THEN
    BEGIN
      j := 0;
      wait;
    END;
  END { CHECKBUFFSIZE };

PROCEDURE initialize;

BEGIN
  j := 0;
  reset(terminal, 'TTY:', '/I/B:8/M:1');
  reset(hostfile);
  eofchar := chr(21); { NAK }
  startchar := chr(21);
  wakeupchar := '!';
END { INITIALIZE };

FUNCTION sync: boolean;

VAR
  k: integer;
  ch: char;

BEGIN
  write(tty, 'TYPE S TO START TRANSFER => ');
  get(terminal);
  ch := chr(terminal MOD 128);
  writeln(tty, ch);
  IF ch IN [ 'S', 's' ] THEN
    BEGIN
      write(tty, startchar);
      FOR k := 1 TO PAUSELEN DO
        sync := true;
    END
  ELSE
    sync := false;
END { SYNC };
BEGIN { DRIVER }
initialize;
IF sync THEN
BEGIN
WHILE NOT eof(hostfile) DO
BEGIN
WHILE NOT eoln(hostfile) DO
BEGIN
read(hostfile, ch);
j := j + 1;
write(tty, ch);
checkbuffsize(j);
END;
readln(hostfile);
j := j + 1;
write(tty, chr(13));
checkbuffsize(j);
END;
FOR i := 1 TO PAUSELEN DO 
write(tty, eofchar);
END ELSE
writeln(tty,
'NO TRANSFER STARTED...TYPE CONTROL-C');
END { DRIVER }.
Vita

Michael Carr was born on December 31, 1955. A native of Chester, New Jersey, he graduated from West Morris Mendham High School in June 1973, and attended Stockton State College in Pomona, New Jersey from 1975 to 1980. After receiving a Bachelor of Science degree in Mathematics from Stockton, he resided in western Massachusetts before attending Lehigh University in Bethlehem, Pennsylvania from 1981 to 1983 as a graduate student in Computing Science.