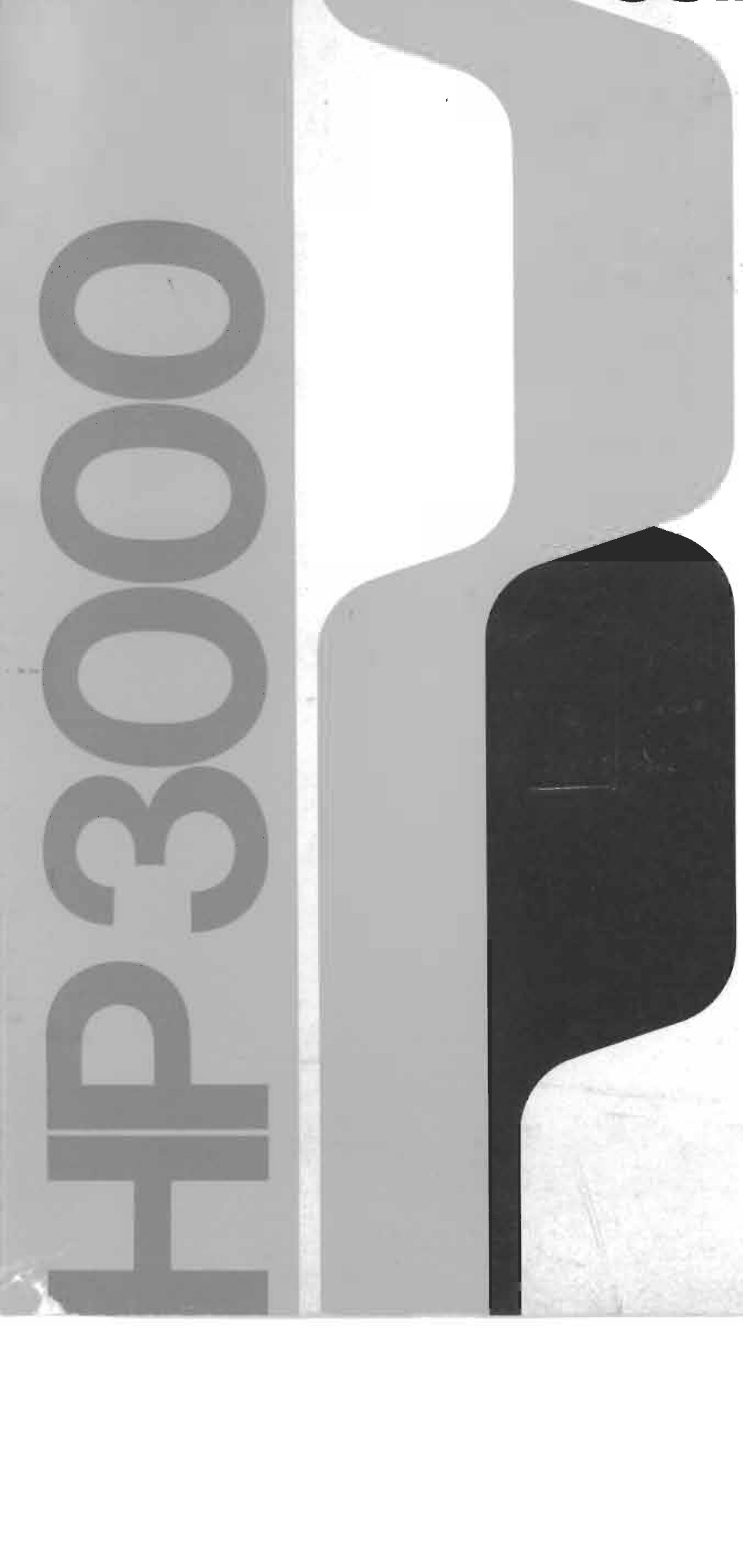
HEWLETT PACKARD

USING THE HP 3000

an introduction to interactive programming



using the HP 3000

an introduction
to interactive
programming

First Edition	Aug 1977
Second Edition	May 1978
Third Edition	Jan 1979
Fourth Edition	April 1979

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another program language without the prior written consent of Hewlett-Packard Company.

HEWLETT hp PACKARD

Part No. 03000-90121

5303 Stevens Creek Blvd., Santa Clara, California 95050

Printed in U.S.A. 4/79

how to use this manual

This manual demonstrates how simple it is to use the HP 3000.

Of course, "simple" to the professional computer programmer does not mean the same thing that it does to someone who has never used a computer before. And we know from experience that both categories are well represented in the HP 3000 user base.

In order to be useful to the widest range of new users, this manual uses a dual approach. That is, we will fully address the beginner and provide short-cuts for the professional.

If you are not an experienced programmer we recommend that you read and do all the Sections 1 through 3, plus all of the particular language section that is of interest to you.

Note: A complete summary of all commands introduced in a particular section is presented in front of that section.

If you are an experienced programmer you should:

a. Skim each Summary in Sections 1 and 2, and read Section 3.

b. Do all operations shown by colored printing in Section 1. (Read supporting text only if you don't understand what you are doing.)

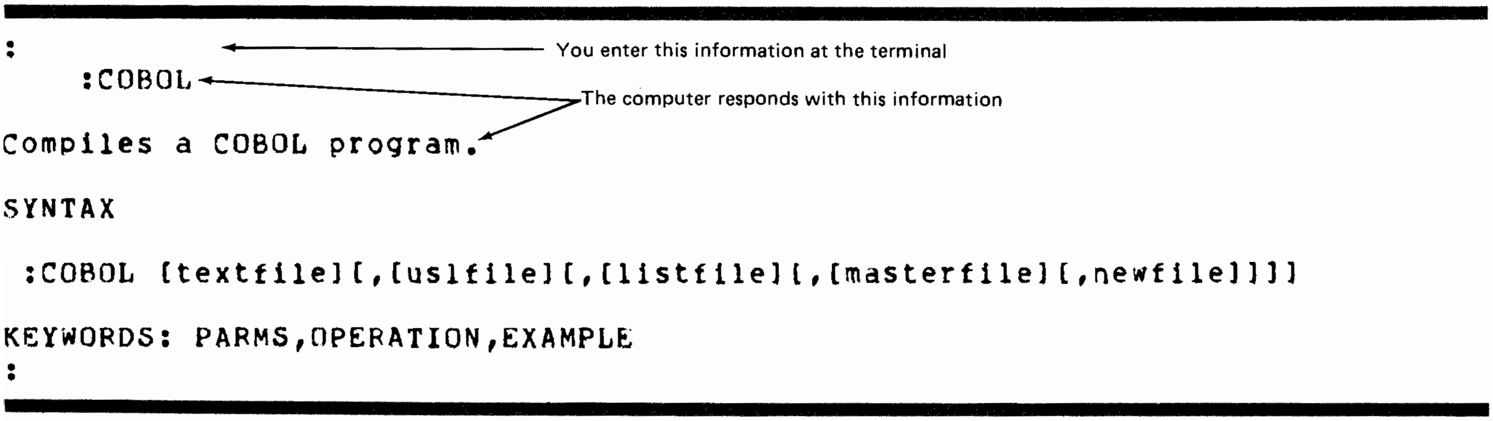
c. Go to the language section that is of interest to you, and do as you did in Sections 1 and 2.

Do not expect this manual to teach you programming. We assume that you have learned some programming language – perhaps on another computer, or in a computer class, or out of a book. You *will* be shown how to get your programs into the HP 3000 and how to get them to run.

You should also be aware that, as an elementary guide, this manual presents only a small subset of all operations that are possible on the HP 3000. Even for the few commands demonstrated, we have purposely omitted many of the powerful but more involved options. So if anything you are shown seems like a limitation or disadvantage, be sure to consult the reference manuals to see how your real-life situations can be handled.

Also, this manual is addressed primarily to users whose HP 3000 operates under the control of the MPE-III operating system. If your system uses MPE-C, however, this manual should still be of use, even though some differences will be obvious.

You are encouraged to do the examples illustrated in this manual at a terminal. To distinguish the information that you type in to the terminal from the information that is displayed to you by the computer, the user inputs are colored in yellow. **For example:**



```
:
     :COBOL

Compiles a COBOL program.

SYNTAX

 :COBOL [textfile][,[uslfile][,[listfile][,[masterfile][,newfile]]]]

KEYWORDS: PARMS,OPERATION,EXAMPLE
:
```

You enter this information at the terminal

The computer responds with this information

contents

Description Page No.

CONDUCTING A SESSION

USING EDIT/3000

Description Page No.

DEVELOPING PROGRAMS

USING FORTRAN/3000

Description **Page No.**

USING COBOL/3000

Description **Page No.**

USING BASIC/3000

USING RPG/3000

section 1
conducting a session

summary of commands introduced in this section

To log on:

a. Turn on the terminal, set to remote mode.

b. If necessary, dial up the computer over telephone lines.

c. Press RETURN. Wait for the colon prompt (:).

d. Type HELLO and log-on identification. Follow with ";TERM=*nn*" if needed. Ask your system manager for the proper term type.

e. Wait for the next colon prompt.

To delete a character, type H^c, or press the backspace key.

To delete a line, type X^c.

:ABORT

terminates a program or operation after the program or operation has been suspended by pressing the BREAK key.

Note: The RETURN key should be the last key pressed for all messages entered at the terminal. The RETURN key is a signal to the computer that a complete message has been entered.

H^c means that 2 keys are used simultaneously. The superscript represents the CNTL key. First press the CNTL key down and hold it down. Now press the H key and then release both keys.

:BYE

terminates your session.

:HELP commandname

lists the formal syntax of any MPE-III command.

:HELP commandname,OPERATION

gives an explanation of the operation of any MPE-III command and all associated parameters.

:HELP commandname, EXAMPLE

gives an example of any MPE-III command.

:HELP commandname, PARMS

lists all parameters of the specified command.

:REDO

allows you to edit the last issued MPE-III command.

conducting a session

There are several ways to talk to an HP 3000 computer. The most common way is through a terminal. Other methods such as punched cards, magnetic tape, paper tape, and even computer-to-computer communications are merely extensions of the principles you will learn here.

Before you come to the terminal you will need your "log-on" identification. Mine is

ED.DATASYS3

meaning that my user name is ED and I want my charges for computer time billed to the DATASYS3 account.

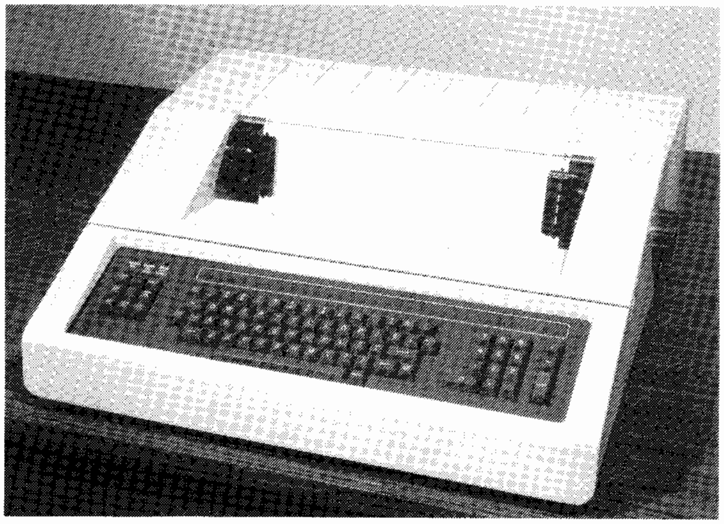
You too must have a user name and an account name – names that your computer will accept as legitimate. Also, if you will be using a telephone connection to the computer, you will need the computer telephone number. Your system manager can give you these.

Included in your log-on identification there may also be passwords. Let the computer ask for them.

In any case, you must have all this information and it must be exact; otherwise your attempt to log on will be rejected.

Your terminal is likely to be a CRT terminal like this . . .

or a hardcopy terminal like this . . .



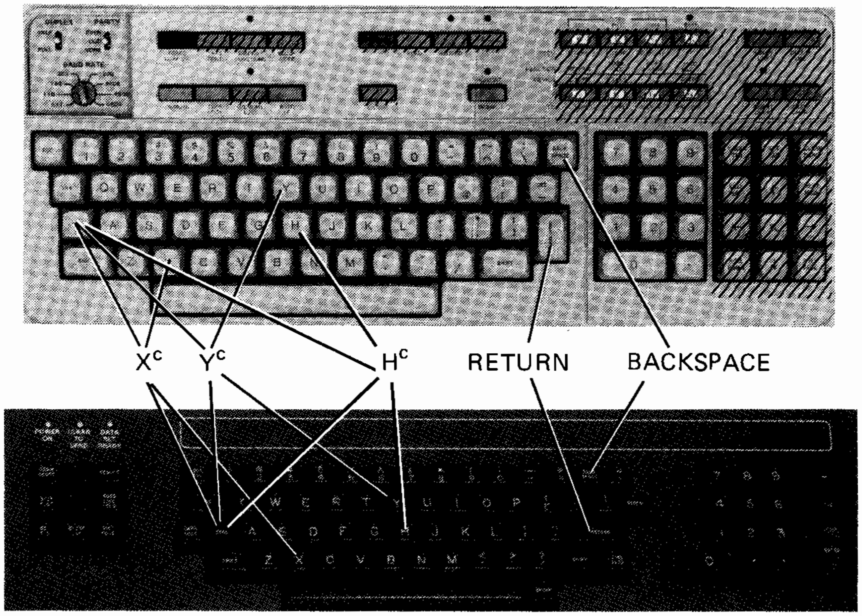
The keyboards of these particular terminals are arranged as shown below. Some of the important keys that we will be discussing shortly are pointed out on these diagrams. Note that the shaded areas on the keyboard should not be used during your session.

The keyboard is your means of talking to the computer. The computer will answer you either by printing on the paper of your hardcopy terminal or by writing on the face of the CRT (cathode-ray tube) of your CRT terminal.

So that you can keep track of what you are typing in, and to be sure that the computer is receiving it correctly, the computer returns (echoes) each character you type in and prints or displays it on the terminal. This happens so fast that it seems you are typing directly to the paper or CRT. In actual fact, though, the computer does all the printing and writing.

As a result of this echoing, the printed or displayed information that you will see on the terminal consists of both what you said to the computer and what it answered back. To distinguish the two types of information in this manual, user inputs are printed in color and computer responses are printed in black.

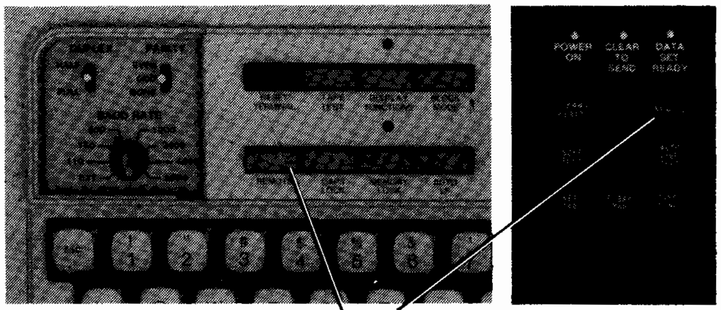
HP 2645A CRT Terminal Keyboard



HP 2635 Hardcopy Terminal Keyboard

now begin...

Turn on the terminal and set the remote mode. *Remote mode* means that all signals generated by pressing keyboard keys are routed out to the computer rather than simply controlling the terminal itself (*local mode*).



Set to remote

If your terminal is like the HP 2645A, it has several control switches which have two positions – (1) up for Off and (2) down for On. Set them as follows:

REMOTE	On	
BLOCK MODE	Off	
AUTO L.F. (Line feed)	Off or On	
CAPS LOCK (Upper case) . . .	Off or On	
DUPLEX	FULL	
PARITY	NONE	
BAUD RATE	2400	(or maximum below 2400, equivalent to 240 charac-
	300	ters per second) for type 103 modem

If your terminal is connected directly to the HP 3000 (i.e., does not need a telephone connection) skip the following paragraph.

If you do need a telephone connection:

a. Dial the computer telephone number.

b. Wait for the computer to return its "carrier signal."

> **Note:** For acoustically coupled telephone equipment this signal will be audible as a high pitched tone in the phone handset; for other equipment a CARRIER light will go on. If you do not get the carrier signal, the phone line may be busy (busy signal heard) or the computer may not be in operation at this time.

c. When the carrier signal is received, place the handset firmly in the receptacle (or, for some equipment, press the DATA button).

your first log-on

a. Press RETURN.

Time Limit:
You must log on within a few minutes after pressing the RETURN key or the computer will disconnect you. The exact time varies from system to system. If this happens when you are calling the computer through a telephone line, redial the computer telephone number.

b. Log on by typing after the colon:

1. the word HELLO
2. a space
3. your log-on identification

For example, to log on through an HP 2645A terminal your log-on would look something like

```
:HELLO ED.DATASYS3
```

c. Press RETURN.

Your pressing the RETURN key means the following things to the computer:

1. You are sure the line you just typed is complete and correct.
2. You want the computer to act on what was typed.
3. You want the computer, after acting, to return the terminal carriage (or CRT cursor) to column 1 of the next line for more commands.

Log-on and Welcome message:
If your log-on is good, the computer responds with a standard log-on message which will look something like this:

```
HP3000 / MPE III.  THU, FEB 23, 1978,  2:59 PM
```

A brief welcome message from your computer operator may also appear. You should note it.
Example:

```
PLEASE PUT ALL UNWANTED LINE PRINTER LISTINGS IN THE
RECYCLE BOX AT THE SIDE OF THE LINE PRINTER.  THANK
YOU, J. BOXER, SYSTEM OPERATOR.
:
```

The final colon indicates that the computer is ready for your next command.

But if you err . . .

The computer will point out that you made a mistake, perhaps like this

```
:HELLO ED,DATASYS3
HELLO ED,DATASYS3
EXPECTED ACCOUNT NAME.  (CIERR 1426)
:
```

or like this

```
:HELLO ED.DATASYS
 HELLO ED.DATASYS
NON-EXISTENT ACCOUNT.  (CIERR 1437)
:
```

meaning, respectively, that your punctuation is incorrect (note erroneous comma) or that your log-on identification is not valid (possibly spelled wrong, or the user name or account name does not exist, or you failed to give the correct password).

But don't take offense. No harm has been done.

The repeated colon means that the computer is giving you another chance. Try again.

If your log-on is successful, the display on your terminal looks similar to the following:

```
HP3000 / MPE III.  THU, FEB 23, 1978,  2:59 PM

PLEASE PUT ALL UNWANTED LINE PRINTER LISTINGS IN THE
RECYCLE BOX AT THE SIDE OF THE LINE PRINTER.  THANK
YOU, J. BOXER, SYSTEM OPERATOR.
```

summary

To log on:

a. Turn on the terminal, set to remote mode.

b. If necessary, dial up the computer over telephone lines.

c. Press RETURN. Wait for the colon prompt.

d. Type HELLO and log-on identification. Follow with ";TERM=*nn*" if needed. Consult your system manager for the proper terminal type.

e. Wait for the next colon prompt.

correcting typing mistakes

Sometime, somehow, you will press the wrong key. We guarantee it.

What then? Well, let's make some mistakes and see.

After logging on, type in any word (but *do not* press RETURN), and deliberately misspell the word. (If you do press RETURN, an error message is printed as explained later; just retype the word.) Suppose in trying to type MISSISSIPPI we have

```
:MISSISXIPPI
```

Comparing this with the correct spelling, we see that we must "back up" five character positions to get back to the point where we went wrong. So . . .

TO CORRECT PART OF A LINE

a. Press the CTRL (or CNTL) key and keep holding it down.

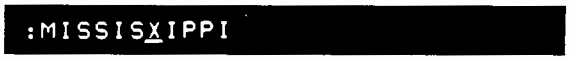
b. Strike the H key once for each character position you want to back up.

Notes: This character combination is called control-H, and written H^c.

On a terminal which has no backspacing capability (most hardcopy terminals), a back-slash will be printed for each H^c, so that your printout will look like this:

```
:MISSISXIPPI\\\\\
```

On most CRT terminals you will notice that the cursor (blinking character position) has backed up five spaces.

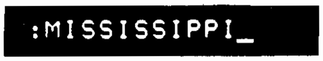


```
:MISSISXIPPI
```

These indications (the back-slashes or backed up cursor) tell you that your last five characters have been deleted from the computer's memory, although they are still visible to you.

c. Now type in the correct characters (SIPPI, in this case). After this you will have

```
:MISSISXIPPI\\\\\SIPPI
```



```
:MISSISSIPPI_
```

On an HP 2645 terminal, there is a backspace key. You can use it rather than H^c to back up to the required position.

TO RETYPE THE WHOLE LINE

If your error is far back in the line, like

```
:NISSISSIPPI
```

(still assuming you have not yet pressed RETURN), you may prefer to scrap the whole line. In this case:

a. Press the CTRL (or CNTL) key and keep holding it down.

b. Strike the X key.

Notes: This character combination is called control-X and is written X^c.

The computer immediately prints out a triple exclamation mark (!!!) and returns the carriage (or cursor) to character position 1 on the next line. This means that the computer has erased that entire line from its memory, although it is still visible to you.

c. Retype the line.

```
:NISSISSIPPI!!!
MISSISSIPPI
```

Notes: If you did actually type this nonsense on your terminal, type X^c now to get rid of it.

BUT IF YOU ALREADY PRESSED RETURN . . .

It is too late. The computer will come back at you with an error message, probably:

```
:NISSISSIPPI!!!
MISSISSIPPI
 ^
UNKNOWN COMMAND NAME.  (CIERR 975)
:
```

The colon indicates that everything is okay and the computer is waiting for a good command from you.

redo and help

Note: These commands are not available on an HP 3000 that uses the MPE-C operating system.

There are two commands available that can save you time, both in correcting improperly entered commands, and in learning a particular command.

The first of these is REDO, which allows you to edit the last command that you enter. To use it, you simply type REDO. After the previous command is echoed, you can insert, delete, or replace characters in it. Use one of these edit symbols. I (for insert), D (for delete), R (for replace), or U (for undo all edits) to change the text of the command. If you do not use an edit symbol I, D, or U, MPE assumes that you want to replace a character.

To illustrate, let's deliberately make an error. Suppose, for example, that you want to see a list of all the users who are currently using the system. The correct command is SHOWJOB, but type SHOWJOV instead:

```
:
 -
UNKNOWN COMMAND NAME.  (CIERR 975)
:
```

Now type REDO and type a B under the V in SHOWJOV when it appears:

```
:
SHOWJOV

SHOWJOB
```

When you press the RETURN key, MPE rewrites the command as you have specified, and waits for you either to make more corrections, or to tell it that you want it to accept the changed command as it now appears. In this case, you want it to accept the modified command, so tell it that you do by pressing the RETURN key again. MPE executes the command, and returns a listing which looks similar to this:

```
JOBNUM   STATE IPRI JIN   JLIST     INTRODUCED   JOB NAME

#S2400   EXEC        83    83       FRI 10:54A   TOM.CLIFTON
#S2394   EXEC        90    90       FRI 10:43A   JOHN,U6.SPL
#S2368   EXEC        26    26       FRI 10:01A   ED.DATASYS3
#S2369   EXEC        47    47       FRI 09:34A   INTRO.BASIC

4 JOBS:
    0 INTRO
    0 WAIT; INCL 0 DEFERRED
    4 EXEC; INCL 4 SESSIONS
    0 SUSP
JOBFENCE= 2; JLIMIT= 6; SLIMIT= 60
```

The extreme right-hand column is a list of all users who are currently using the system. Your log-on identification should appear in this column. Some of the other entries are clear, such as your log-on time, and the state in which your session currently resides. Other information given in the table is not discussed here, as it is not appropriate. However, if you wish more information, refer to the SHOWJOB command in the *MPE Commands Reference Manual.*

The second command, HELP, can be used to give you information on any MPE-III command. It has three optional parameters (called keywords), which are PARMS, OPERATION, and EXAMPLE. To illustrate, let's take a look at the LISTF command, using the OPERATION parameter. Enter the following command:

```
:HELP LISTF,OPERATION
OPERATION
```

Your terminal should display the following message:

```
Lists descriptions of one or more disc files at level of detail you
select.  You need not have access to a file to list a description of it,
however,. a file description will not be listed unless the file's home
volume set is mounted.  A standard user may list level 0, 1, and 2
information for any file in the system.  A user with Account Manager
capability may list level -1 data for files in his own account. A user
with system Manager capabilities may list data for any file in the
system.

Note that this command applies only to permanent disc files in the
system file domain.

KEYWORDS: PARMS,OPERATION,EXAMPLE
```

If you want to see the syntax, a list of the parameters, or an example, you enter HELP LISTF, or HELP LISTF,PARMS, or HELP LISTF,EXAMPLE, respectively.

Note that if your system is not operating under the control of MPE-III, these commands do not exist, and an attempt to use them results in an error message.

your first log-off

To end your session with the HP 3000, all you have to do is to type BYE when the computer prompts you for a command, and then press the RETURN key.

Your friendly HP 3000 signs off, telling you how long it spent working for you (CPU time, in seconds), how long your session lasted (connect time, in minutes), and the date and time that you ended the session. It then disconnects from your terminal. For example:

```
CPU=26.   CONNECT=42.   THU, FEB 23, 1978,   3:41 PM
```

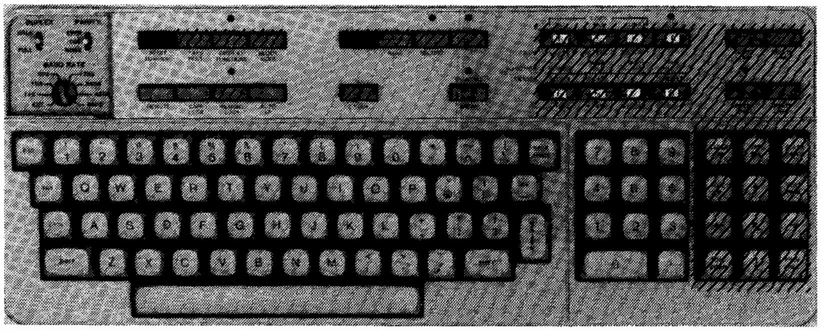
At this time the CARRIER light (if your terminal has one) will go off, meaning that you are disconnected from the computer. Direct-connected terminals will show no indication of disconnection after the end of session message.

The terminal may now be turned off. The handset, if your terminal uses one, should now be returned to the telephone cradle.

AND AS A FINAL NOTE . . .

The shaded areas on some terminals such as the HP 2645A pictured below, should not be used. They are not suitable for the mode being used here (remote operation, character mode).

HP 2645A CRT Terminal Keyboard



summary

To delete a character, type H^c, or press the backspace key.

To delete a line, type X^c.

:BYE

terminates your session.

:HELP commandname

lists the formal syntax of any MPE command.

:HELP commandname,OPERATION

gives an explanation of the operation of any MPE command and all associated parameters.

:HELP commandname,EXAMPLE

gives an example of any MPE command.

:HELP commandname,PARMS

lists all parameters of the specified command.

:REDO

allows you to edit the last issued MPE command.

introducing the interactive subsystems

Up to this point, you have been dealing directly with the computer's operating system. Rarely, however, will you spend much time communicating directly with the system. Usually all you will do in this mode (which we will call *system mode*) is specify a data file, eliminate a data file, or request the compilation or execution of your program. And possibly you may do none of these in system mode.

Instead, you will do most of your communicating with one of the operating system's *interactive subsystems.* The two most common are EDIT/3000 (Text Editor) and the BASIC/3000 Interpreter.

You will need to become familiar with one or the other of these subsystems, depending on the language you intend to use. If you intend to use only BASIC, you do not need to learn the Editor. If you intend to use any other language (FORTRAN, COBOL, RPG, SPL), you will need to learn the Editor.

just in case...

When you are running a program, or working within a subsystem, and you get into a situation that you feel entirely uncomfortable with, DON'T PANIC!

There is nothing you can do from a terminal that will harm the computer in any way, and there is a way to remove yourself from an uncomfortable situation. It consists of two steps, which are:

a) Press the BREAK key. This tells the computer that you want it to temporarily stop whatever it is doing for you and prompt you for a command.

b) Type ABORT after the computer responds with a colon prompt. This stops your program completely, and, if you were in a subsystem, it gets you out of it. When this occurs, the error message, PROGRAM TERMINATED PER USER REQUEST. (CIERR 989), is printed. This simply informs you that the program did not terminate itself, but instead, that MPE had to do it.

You can now re-enter the subsystem, make changes to the program, and attempt to run it again.

section 2
using EDIT/3000

summary of commands introduced in this section

:EDITOR	gets you into the Editor subsystem
/ADD	allows you to type text into the work area
/ADD 3.1	enters new lines between lines that already exist; in this case, between 3 and 4
/ADD 5.1, HOLDQ,NOW	inserts hold file contents after line 5
Y^c	terminates the add mode
/DELETE 3	deletes line 3 (of work area)
/DELETE 4/15	deletes lines 4 through 15
/DELETE ALL	clears work area, if you answer YES to CLEAR?
/END	wipes out your work area and gets you out of the Editor subsystem
/GATHER 11/15 TO 8.1	moves lines 11 through 15 to 8.1 through 8.5
/GATHER ALL	renumbers all lines; increment of 1, start at 1
/HOLD ALL	stores work area in hold file
/HOLDQ 9/33	stores only lines 9 thru 33
/JOINQ FILE6	joins contents of FILE6 to end of work area
/KEEP FILE6	stores work area in FILE6
/KEEP FILE6 (9/33)	stores only lines 9 thru 33
/MODIFY 2	prints out line 2 and waits for your modifications (next 3 items); use RETURN to terminate

```
ABCDEFGHIJK      original line
      D          deletes character above D
ABCDEFHIJK       modified line

ABCDEFGHIJK      original line
 D  D            deletes characters above
AFGHIJK            and between pair of D's

ABCDEFGHIJK      original line
   IXYZ          inserts characters preceding
ABCXYZDEFGHIJK     character above I
```

/TEXT filename	brings contents of disc file into work area
/LIST ALL	prints the contents of your work area
/LIST 1/7	prints lines 1 through 7
/LIST 2	prints line 2
/LIST ALL,OFFLINE	prints work file at the line printer (you may need a FILE equation)
:PURGE filename	eliminates an old file
:LISTF	lists all existing file names in the file group which contains your files; can terminate listing with BREAK key
:FILE LP;DEV=deviceclass name	used to create a file reference for the line printer.

using EDIT/3000

EDIT/3000 is the basic tool for interactively creating a source file on the HP 3000.

It can be viewed as a blackboard upon which you write your source file, allowing you to erase parts, insert other parts, modify others, or even to copy a file onto or off of it.

The discussion which follows is only an introduction to the Editor and its capabilities, but is more than sufficient for the purposes of this manual. If you wish more information about EDIT/3000, refer to the *EDIT/3000 Reference Manual* (part number 03000-90012).

If you are an experienced programmer, you may want to skim this section, and study the summary of commands which are introduced in this section; all are listed on page 2-3.

FOR BASIC USERS

This section of the manual is concerned with the Editor and simple file operations. Since the BASIC Interpreter has its own editing and file facilities, you may want to skip from here directly to the BASIC section.

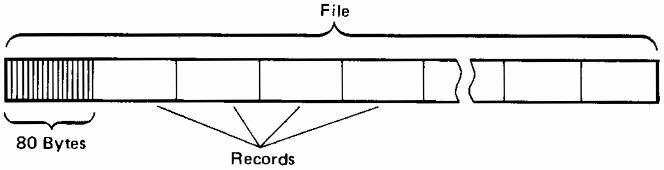
source files

The primary purpose of the Editor is to accept your program statements, typed in at the terminal, and form them into a *source file* on disc.

After a program source file has been created, it is a simple matter to compile, prepare, and run it (this is done outside of the Editor subsystem).

So our concern now is how to create source files. Later, starting on page 2-11, we will discuss a secondary purpose of the Editor, which is the editing of source files. Still later, in the succeeding sections, you will learn about compiling, preparing, and running programs in various languages. But one thing at a time . . .

First, let's visualize what a source file looks like on disc. A source file is made up of records, which, in turn, are made up of bytes:



A byte is a sequence of eight bits (binary integers) operated on as a unit by the computer.

Each byte can store one character that you type in via the terminal. Each record consists of 80 bytes and can therefore store one line of typing, up to 80 characters. (You will rarely fill up the full record, although the storage space is always there.) Each file can be made long enough to contain as many line-records as you need. The Editor will automatically create a file of the correct size when you issue a KEEP command, and can later adjust the length according to your needs.

A file might have contents or it might be empty. In any case, it *always* has a name. You choose the name when you create the file and you mention the same name whenever you do something with that file. The name can be one to eight alphanumeric characters, beginning with a letter.

The computer stores all files created under your user name in various places on the disc. However, the computer maintains a file directory where all your files are listed in a *file group*. So when, for example, you want the computer to get one of your files for you, it first looks in its file directory for your file group and then for the name of the file you want. Noting the disc address of your file, the computer easily picks up the contents of the file and does with it whatever you specify.

This is essentially what happens when you use the Editor to list (print out) a disc file (as we will do very soon). You tell the computer to LIST the file. The computer finds the file and copies it out on your terminal, one record at a time.

Now let's actually try some elementary file operations and see how it's done.

HOW TO CREATE A DISC FILE

If you are not currently logged-on, log on now and wait for the colon prompt.

You get into the Editor by typing EDITOR in response to a colon prompt from the operating system. (But don't do any typing just yet.)

Only the operating system uses a colon for its prompt character. The prompt character for Editor commands is /.· If you unexpectedly get a colon instead, it is because you accidentally pressed the BREAK key. In that case type RESUME to get back into the subsystem.

Now it's time to enter the Editor subsystem. Type EDITOR and wait for the slash prompt.

```
:
HP32201A.7.01 EDIT/3000 TUE, FEB 27, 1978 3:08 PM
(C) HEWLETT-PACKARD CO. 1976
/
```

You are now talking to the Editor. Without saying so, the Editor has prepared a work area for your exclusive use. This work area has room for about 2000 line-records.

You can write anything you want in this work area. All you do is go into the Editor's "add" mode. This is done simply by typing ADD after the slash prompt. After you press RETURN, the terminal prints "1" (meaning line 1), spaces over five character positions, and waits for you to begin typing. Your next RETURN will cause a 2 to be printed (meaning line 2). And so on. Try entering four lines. Use H^c or X^c to correct any typing errors. (If you have to use X^c, start typing in column 1 after the carriage return; don't try to indent.)

```
/
     1
     2
     3
     4
     5
```

To get out of the add mode, either type two slashes (//), or press CTRL (or CNTL) and then strike the Y key. This control-Y (Y^c) combination terminates the ADD command, prints out three periods, and prints a new slash prompt for a new command.

So now you have four records in your work area. You can always check what you've got in your work area by typing LIST ALL after a slash prompt.

```
     5        ...
/
```

```
/
     1        *** ENTER: THE GHOST OF CAESAR ***
     2        HOW ILL THIS TAPER BURNS! HA! WHO COMES HERE?
     3        I THINK IT IS THE WEAKNESS OF MINE EYES
     4        THAT SHAPES THIS MONSTROUS APPARITION.
/
```

If you are now satisfied that the contents of your work area are correct and worth keeping, and you want to put the contents into a permanent disc file, type:

```
/
```

The Editor now proceeds to create a file named BRUTUS39 and stores the contents of your work area into that file. This may take a few seconds. When you get a new slash prompt you know that the file exists and contains a copy of whatever is in your work area.

EDITOR ERRORS

If you do something wrong while using the Editor, the Editor may give you an immediate explanation or it may just print out an error number.

For example, if you already have something in the file you are now attempting to store into, you will get a message that looks like this:

```
BRUTUS39 ALREADY EXISTS
     - RESPOND YES TO PURGE OLD AND THEN
KEEP PURGE OLD?
```

You can try this if you want to; just type another KEEP BRUTUS39.

You can answer YES to purge the old file and replace it with a new file of the same name which has the new contents.

Try it; type YES. This will just copy the same unchanged contents of your work area into the new BRUTUS39 file.

Alternatively, you can answer anything else (e.g., NO or a carriage return) to cancel the current command.

Try that if you want; retype KEEP BRUTUS39 and answer NO to the PURGE OLD? question. The Editor then prints PURGE OF OLD FILE NOT CONFIRMED – TEXT IS NOT KEPT, followed by a new slash prompt. This gives you a chance to rectify your error. Assuming that you want to keep *both* the old file and the new one, you will probably want to specify a new file. To do that, type a new KEEP command with a different file name; e.g.,

```
                    /
```

As another example, suppose you neglected to specify the file name in a TEXT command. (We'll be discussing the TEXT command on the next page.) You would get an error message that looks like this:

```
*  3*
```

After printing out this message, the carriage or cursor stops on the same line. You can either press RETURN or, to get an expanded explanation of your error, type any character; for example a dash:

```
*  3*
MISSING PARAMETER
/
```

You may now retype the command correctly, including the required parameter.

Still another possible error you may encounter (for example, if you neglect to specify the file name in a KEEP command) is one that prints out a File Information Display block prior to the error number (*60*). This display block provides 14 lines of information about the file and the conditions of failure. For now you don't need to be concerned about the contents of the display. Just press any printing key after the error number is printed out so you will get a short explanation of the error.

GETTING THE FILE CONTENTS BACK

So your data is now in a disc file. How do you get it back?

You can bring the contents of any of your files back into the work area by using a TEXT command. Type TEXT and the file name, as follows:

```
/TEXT BRUTUS39
```

If there is presently anything in your work area (which will be the case if you've been following these directions on your terminal), you will be asked if it is okay to clear your work area. This is to remind you that the TEXT command erases any work area which has been altered before bringing in a file. Usually you will answer YES as shown below. Otherwise, the TEXT command is cancelled. The normal printout is:

```
/TEXT BRUTUS39
IF IT IS OK TO CLEAR RESPOND "YES"
CLEAR? YES
/
```

A copy of the file BRUTUS39 is now in your work area. You can confirm this by typing LIST ALL.

```
/LIST ALL
     1        *** ENTER: THE GHOST OF CAESAR ***
     2        HOW ILL THIS TAPER BURNS! HA! WHO COMES HERE?
     3        I THINK IT IS THE WEAKNESS OF MINE EYES
     4        THAT SHAPES THIS MONSTROUS APPARITION.
/
```

Now let's say you want to add a line to what you've already got. Again type ADD. Notice that your prompt is now "5", which is the next line number in sequence. Type in your new line. Example:

```
/
     5
     6
```

Again, to terminate the ADD command, type two slashes or press Y^c.

When you again list the contents of your work area you will see the new line added in correct sequence.

```
/LIST ALL
    1       *** ENTER: THE GHOST OF CAESAR ***
    2       HOW ILL THIS TAPER BURNS! HA! WHO COMES HERE?
    3       I THINK IT IS THE WEAKNESS OF MINE EYES
    4       THAT SHAPES THIS MONSTROUS APPARITION.
    5       IT COMES UPON ME. ART THOU ANY THING?
/
```

Since this is a new and better version of what we had before, let's save it as the new BRUTUS39 file. We don't care about the contents of the old BRUTUS39, so respond YES when asked if you want to purge it.

```
BRUTUS39 ALREADY EXISTS
        - RESPOND YES TO PURGE OLD AND THEN KEEP
PURGE OLD?YES
/
```

This concludes our introduction to files. If you want to continue with more instruction in this session, skip the next four paragraphs.

If you want to stop for a while at this point, type END to get back to system mode.

```
/END
 END OF SUBSYSTEM
:
```

To avoid cluttering your account with experimental files, eliminate the files you've been using in this session. Use the PURGE command as shown below. If you don't know the file names of the files you want to purge, or if you are told FILE NOT FOUND, type LISTF. This command lists all the files in your file group (which may include files of other users using the same account). If the list runs on too long, you can terminate it at any time with the BREAK key.

The LISTF and PURGE sequence looks like this:

```
:LISTF

FILENAME

BRUTUS39
:PURGE BRUTUS39
:
```

The next colon prompt signifies that your command has been carried out. You may now log off if you want to.

summary

:EDITOR	gets you into the Editor subsystem
/ADD	lets you type text into the work area
// or Y^c	terminates the add mode
/LIST ALL	prints the contents of your work area
/KEEP filename	stores your work area into disc file
/TEXT filename	brings contents of discfile into work area

Any printing key pressed after an error number provides an expanded explanation of the error; press RETURN if explanation is not needed.

/END	wipes out your work area and gets you out of the Editor subsystem
:PURGE filename	eliminates an old file
:LISTF	lists all existing file names in the file group which contains your files; can terminate listing with BREAK key.

editing your programs

As we all know, no program is perfect as originally written. You will want to modify and insert lines, as well as just add lines as we did previously. You may also want to set aside whole blocks of code and later join them together. All of these facilities (and several more) are provided by the Editor.

Some of these basic editing operations will now be shown. In order to remain very general in this discussion, we will use a model program in outline form rather than an actual program in some specific language.

Let's begin by taking a look at the model program shown in the next column.

The first thing you must get used to is indenting accurately. Very few terminals tell you the column number for the current position of the carriage or cursor. So you will have to do the counting yourself.

You already know that the ADD command automatically spaces over several character positions (10 actually). Those positions are reserved for the line number. Where the carriage or cursor stops is where you begin typing in your program; that position is "column 1."

The model program, as you can see, uses two levels of indention. (Later we will have a third level.) The degree of indention chosen here is in multiples of four. So rather than counting out eight or twelve spaces, learn to strike the space bar in groups of four–once for the first level of indention, twice for the second level, three times for the third.

This is important to remember after you do a X^c. Since the Editor already has the line number, the carriage or cursor returns to the extreme left and waits for your new input. Thus visual alignment of text is impossible. You must count out your indention from the extreme left position.

Okay? Now type the following. (If necessary, first log on and type EDITOR.)

```
/
         1
         2
         3
         4
         5
         6
         7
         8
         9
        1Ø
        11
        12
        13
        14
        15
```

Type // or Y^c to terminate. If you had to use H^c or X^c several times, do a LIST ALL to get a clean copy in front of you. If minor errors still exist, leave them. If you want to start all over, type DELETE ALL (and answer YES when asked if it is okay to clear to the work area) and try again.

DELETING ONE LINE

Type DELETE followed by the line number of the line you want deleted. Example:

```
/
         3          COMMENT LINE 1
/
```

Notice that the deleted line is printed out. This is for your protection. If by accident you specify a large number of lines for deletion (see next paragraph) and you observe that the wrong lines are being deleted, you can press Y^c to prevent further destruction of the program.

DELETING SEVERAL LINES

Type DELETE followed by the beginning and ending line numbers of the lines you want deleted, separated by a slash. And if you are confident about the range being correct, add a "Q" to the word DELETE (i.e., DELETEQ). Q stands for quiet, meaning that the deleted lines are *not* printed out; this can save considerable time (and paper if you are using a teletype terminal) if a large number of lines are to be deleted.

```
/DELETEQ 4/5
NUMBER OF LINES DELETED = 2
/
```

Notice that, as a result of the Q, only a total of lines deleted is given.

Another point you should know is that several ranges of line numbers can be given, separated by commas; we could have specified DELETE 2,4/5 and lines 2, 4 and 5 would have been deleted.

LISTING ONLY A PART OF A PROGRAM

To verify that a given edit operation has been performed correctly, you can type LIST followed by the beginning and ending line numbers of the lines you want to examine. Separate the line numbers with a slash. For example, to verify the deletion of lines 3 through 5, list lines 1 through 7.

```
/[illegible]
     1       INITIALIZING COMMAND
     2       PROGRAM NAME9
     6           DECLARATION LINE 1
     7           DECLARATION LINE 2
/
```

To examine only one line (e.g., line 2) you could type LIST 2.

INSERTING LINES

As you've seen, the ADD command normally increments the line number by 1 for each added line. However, smaller increments of .1, .01, or .001 are available. If, for example, you want to add some lines (fewer than 10) between existing lines 3 and 4, you would type ADD 3.1; you will then automatically be prompted for lines 3.1, 3.2, 3.3, etc.

Similarly, if you want to add fewer than 100 lines (but more than 10), you would type ADD 3.01; prompts are then 3.01, 3.02, 3.03, etc. Typing ADD 3.001 works similarly for inserting an excess of 100 lines.

As an example, type the following, *being sure* to indent by striking the space bar four times before beginning to type on each line. (See final result on page 2-12.)

```
/ADD 3.1
    3.1          COMMENT LINE ONE
    3.2          COMMENT LINE TWO
    3.3          COMMENT LINE THREE
    3.4
```

Use Y^c to terminate.

MOVING LINES

It often happens that you have several lines of code that are in the wrong place in your program. To avoid massive retyping, you can simply move the entire block to the correct place.

Suppose, in our model program, lines 11 and 12 should have come after line 8. That is, you want lines 11 and 12 to become 8.1 and 8.2 (applying the principles for inserting lines as described above). To do this, type GATHER 11/12 to 8.1. As each line is moved, the old and new line numbers are printed out.

```
/
    11         =>         8.1
    12         =>         8.2
/
```

You can avoid the old/new line number listing by typing GATHERQ instead of GATHER.

It is likely, though, that you will want to see the results of the move. List the affected area.

```
/
     8                  PROGRAM  STATEMENT  1
     8.1                PROGRAM  STATEMENT  4
     8.2                PROGRAM  STATEMENT  5
     9                  PROGRAM  STATEMENT  2
    10                  PROGRAM  STATEMENT  3
    13                  PROGRAM  STATEMENT  6
/
```

JUST FOR FUN . . .

Let us say that we have just discovered that our program will not do the job intended.

However it will, we note, be useful as a subroutine in a larger program. (You know how it goes.) So we set about changing a few lines to convert the program to a subroutine.

DELETING CHARACTERS

Let's say we want to delete the word PROGRAM in line 2. Type MODIFY 2 and notice that the command is then repeated, line 2 is printed out, and the carriage or cursor returns to column 1 of the next line.

```
/
MODIFY       2
PROGRAM  NAME9
```

This signifies that the Editor is waiting for your modifications for line 2. The modification is to delete the word PROGRAM, so type a D under the first letter of that word and a second D under the last letter of that word. Then press RETURN. Note that the revised line, with PROGRAM deleted, is printed out. Also note that the space between the original two words is still present (at the start of the line); blanks count as real characters.

```
PROGRAM  NAME9

 NAME9
```

You should also be aware that you can delete a single character with one D under the particular character. The remaining characters always shift left to close up the space formerly occupied by deleted characters.

The Editor will remain in the "modify mode" until you are satisfied with the line. That is, you can keep on making modifications to line 2, because the carriage or cursor always returns to column 1 after printing out the modified line, and waits.

To get out of the modify mode, just press RETURN once more. Do that now; that gets us a new slash prompt.

INSERTING CHARACTERS

To insert characters, MODIFY is again used, with an I (instead of a D) typed under the character where the insertion is to begin. For example, to insert SUBROUTINE in front of the blank we left in line 2, type MODIFY 2, type an I under the initial blank character, and type the word SUBROUTINE. That will look like this:

```
/
MODIFY       2
 NAME9

SUBROUTINE NAME9
```

The last line is the final result. Press RETURN once more to get out of the modify mode.

DELETING AND INSERTING CHARACTERS

Replacement of characters can be achieved in two successive operations under the same MODIFY command. First delete the unwanted characters; then insert the new ones; then terminate the modify mode. Example:

```
/
MODIFY     14
END PROGRAM NAME9

END NAME9

END SUBROUTINE NAME9
```

The last line is the new line 14. Press RETURN once more to get out of the modify mode.

Don't stop here, though. You will need the current contents of your work area for the next demonstration. To see what the subroutine looks like at this time, do a LIST ALL.

```
/
     1      INITIALIZING COMMAND
     2      SUBROUTINE NAME9
     3.1        COMMENT LINE ONE
     3.2        COMMENT LINE TWO
     3.3        COMMENT LINE THREE
     6          DECLARATION LINE 1
     7          DECLARATION LINE 2
     8              PROGRAM STATEMENT 1
     8.1            PROGRAM STATEMENT 4
     8.2            PROGRAM STATEMENT 5
     9              PROGRAM STATEMENT 2
    10              PROGRAM STATEMENT 3
    13              PROGRAM STATEMENT 6
    14      END SUBROUTINE NAME9
/
```

JOINING FILES

An entire block of code stored in a file may be joined to the contents of your work area—either at the end of the work area contents or between two consecutive lines. Thus, with parts of programs stored in various files, you can combine them together in almost any arrangement imaginable.

Besides being able to join permanent files, you also have access to a secondary work area called a *hold file.* Like the main work area, its contents are cleared when you leave the subsystem.

To illustrate how both permanent and hold files are used in joining operations, let's put some contents in both. First, the permanent file.

As you did before (page 2-7), specify a disc file and keep the current contents of your work area (the subroutine) in that file.

```
/
/
```

The range specified for the KEEP command deliberately omits line 1 since that line is not part of the subroutine.

Now clear the work area with a DELETE ALL.

```
/
/
```

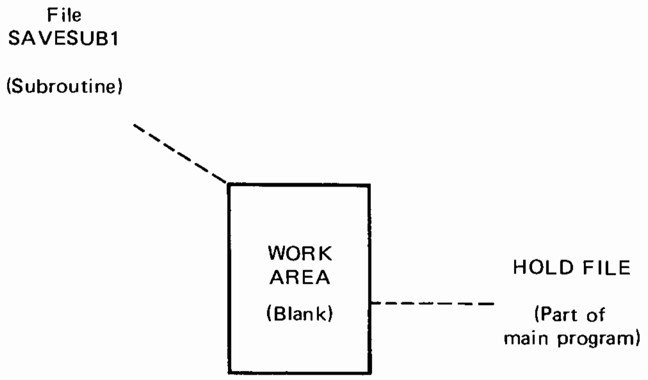
To demonstrate the use of the hold file, imagine that you have just coded the main body of the main program. You now want to get it into the computer and temporarily set it aside. Type the following, *being sure* to indent 8 spaces (two groups of 4). Terminate with Y^c at line 6.

```
/
     1
     2
     3
     4
     5
     6       ...
/
```

Then store this in the hold file with a HOLDQ command as shown below. (The Q avoids listing each line.)

```
/
HOLD FILE LENGTH IS 5 RECORDS
/
```

Now do another DELETE ALL to clear the work area. (HOLD does not affect the work area.) At this time you have part of your main program in the hold file, and a subroutine in a permanent file called SAVESUB1.



Now, in your work area, you put together the remaining elements that will make a complete program: an initializing command, an entry point, some comments, any missing declarations, subroutine calling and return statements, statements to process returned parameters, a program end statement, and so on. To continue with the example, type the following, carefully indenting 4, 8 or 12 spaces where required, using the method previously described.

```
/
     1
     2
     3
     4
     5
     6
     7
     8
     9
    10
    11         ...
/
```

Now begins the joining. Assume we desire to insert the statements now in the hold file at the point between lines 5 and 6 of the current work area. Type:

```
/                     ,NOW
```

(The Q avoids listing each line as it is added into the work area.) The result will look like this.

```
LAST LINE =       5.5
/
```

That takes care of the hold file. (Incidentally, the hold file still contains the original copy of the material just added; if you try to put something else in the hold file, you will be asked if you want to clear it.) Now let's assume that we desire to append the contents of the permanent file SAVESUB1 to the program as it now exists. Type JOINQ SAVESUB1; the result is:

```
/
NUMBER OF LINES JOINED = 13
/
```

At this point, the program should be complete, Let's take a look.

```
/
    1       INITIALIZING COMMAND
    2       PROGRAM MAIN
    3           COMMENT A
    4           COMMENT B
    5           DECLARATION LINE
    5.1             PROGRAM STATEMENT 1
    5.2             PROGRAM STATEMENT 2
    5.3             PROGRAM STATEMENT 3
    5.4             PROGRAM STATEMENT 4
    5.5             PROGRAM STATEMENT 5
    6                   SUBROUTINE CALL
    7               PROGRAM STATEMENT 6
    8               PROGRAM STATEMENT 7
    9           RETURN STATEMENT
   1Ø       END MAIN PROGRAM
   11       SUBROUTINE NAME9
   12           COMMENT LINE ONE
   13           COMMENT LINE TWO
   14           COMMENT LINE THREE
   15           DECLARATION LINE 1
   16           DECLARATION LINE 2
   17               PROGRAM STATEMENT 1
   18               PROGRAM STATEMENT 4
   19               PROGRAM STATEMENT 5
   2Ø               PROGRAM STATEMENT 2
   21               PROGRAM STATEMENT 3
   22               PROGRAM STATEMENT 6
   23       END SUBROUTINE NAME9
/
```

If you prefer to have the lines renumbered to a uniform increment of one (1 through 28 in this case), you could type GATHER ALL.

To save this final copy in a permanent file, we can do another KEEP.

```
/
/
```

The completed, joined program is now in the permanent disc file FINAL.

off-line listings

A useful feature of the Editor is the capability of getting off-line listings of the text you have been editing in your work area. To get an off-line listing, you might have to specify a FILE command before accessing the Editor. If, however, the device class of your line-printer is "LP," only the /LIST ALL, OFFLINE Editor command is needed. Example (assuming device class is "LP1"):

```
            :
```

Then access the Editor in this form:

```
            :
```

rather than just :EDITOR. Then, at any time during this particular access of the Editor, you may give the following command to list the complete work area off-line:

```
            /
```

AND THAT'S ALL you need to know about the Editor for most purposes. You know how to get your source programs into disc files and how to do the most common editing operations. Usually that is all you will need.

leaving the editor subsystem

Type END after any slash prompt to return to system mode.

```
/

 END OF SUBSYSTEM
:
```

If you have created any permanent files that are of no further value, purge them now.

```
:
:
:
```

You may wish to log off now. If so, and you do not know how, refer to page 1-13.

summary

/DELETE 3	deletes lines 3 (of work area)
/DELETE 4/15	deletes lines 4 through 15
/DELETEQ 4/15	deletes without printing deleted lines
/DELETE ALL	clears work area, if you answer YES to CLEAR?
/ADD 3.1	allows 10 lines to be inserted after line 3; use Y^c to terminate
/ADD 3.01	allows 100 lines to be inserted after line 3
/ADD 3.001	allows 1000 lines to be inserted after line 3
MODIFY 2	prints out line 2 and waits for your modifications (next 3 items): use RETURN to terminate

```
   ABCDEFGHIJK        Original line
         D            deletes character above D
   ABCDEFHIJK         modified line

   ABCDEFGHIJK        original line
    D  D              deletes characters above
   AFGHIJK            and between pair of D's

   ABCDEFGHIJK        original line
      IXYZ            insterts characters preceding
   ABCXYZDEFGHIJK       character above I
```

/LIST 1/7	prints lines 1 through 7
/LIST 2	prints line 2
/GATHER 11/15 TO 8.1	moves lines 11 through 15 to 8.1 through 8.5
/GATHERQ 11/15 TO 8.1	does same without listing old/new numbers
/GATHER ALL	renumbers all lines; increment of 1, start at 1
/KEEP FILE6	stores work area in FILE6
/KEEP FILE6 (9/33)	stores only lines 9 thru 33
/HOLD ALL	stores work area in hold file
/HOLDQ	does same without printing lines stored
/HOLDQ9/33	stores only lines 9 thru 33
/ADD 5.1, HOLDQ,NOW	inserts hold file contents after line 5
/JOINQ FILE6	joins contents of FILE6 to end of work area

summary of commands introduced in this section

:BASICOMP MYPROSRC :COBOL MYPROSRC :FORTRAN MYPROSRC :RPG MYPROSRC :SPL MYPROSRC :SPL MYPROSRC	Compiles the source file, MYPROSRC, into a User Subprogram Library file. In this form of the command, the USL file is system file, $OLDPASS. The file, $OLDPASS, is session-temporary. The proper command to use depends on the programming language that the source file is written in.
:BASICOMP MYPROSRC,MYPROUSL :COBOL MYPROSRC,MYPROUSL :FORTRAN MYPROSRC,MYPROUSL :RPG MYPROSRC,MYPROUSL :SPL MYPROSRC,MYPROUSL	Same operation as in the above commands, except that the USL file is the permanent file, MYPROUSL, instead of $OLDPASS.
:PREP $OLDPASS,$NEWPASS	Prepares the object code in $OLDPASS, placing it in $NEWPASS. When preparation is complete, the system file, $NEWPASS, is closed, $OLDPASS is purged, and $NEWPASS is renamed $OLDPASS.
:PREP $OLDPASS, MYPROPRG	Prepares the object code in $OLDPASS, placing the program file in session-temporary file, MYPROPRG.
:SAVE MYPROPRG	Makes the session-temporary file, MYPROP.RG, a permanent file.
:RUN $OLDPASS	Executes the program file, $OLDPASS.
:RUN MYPROPRG	Executes the program file, MYPROPRG.
:BASICPREP MYPROSRC :COBOLPREP MYPROSRC :FORTPREP MYPROSRC :RPGPREP MYPROSRC :SPLPREP MYPROSRC	First compiles the source file, MYPROSRC, into $OLDPASS, then prepares the object file. The resulting file is called the program file, and can be executed. The command used depends on the programming language used in the source file.

summary of commands

```
:BASICPREP MYPROSRC,MYPROPRG
:COBOLPREP MYPROSRC,MYPROPRG
:FORTPREP MYPROSRC,MYPROPRG
:RPGPREP MYPROSRC,MYPROPRG
:SPLPREP MYPROSRC,MYPROPRG
```

Same as the above commands, except that the program file is the session-temporary file, MYPROPRG.

```
:PREPRUN $OLDPASS
```

Prepares the object file, $OLDPASS, placing the prepared code into $NEWPASS. When preparation is complete, $NEWPASS is closed, $OLDPASS is purged, and $NEWPASS is renamed $OLDPASS. $OLDPASS is then executed.

```
:PREPRUN MYPROUSL
```

Prepares the object file, MYPROUSL, using $NEWPASS and $OLDPASS. $OLDPASS is then executed.

```
:BASICGO MYPROSRC
:COBOLGO MYPROSRC
:FORTGO MYPROSRC
:RPGGO MYPROSRC
:SPLGO MYPROSRC
```

Compiles, prepares, and executes source file, MYPROSRC.

The command used depends upon the language that the source file is written in.

developing and running a program

Let's take a look at the commands used to compile, prepare, and execute a program, and the steps the compilers and the segmenter go through in performing these three functions.

COMPILATION AND PREPARATION — A BRIEF DISCUSSION

When you create a source file, you generally use a programming language that is, in itself, unintelligible by the system. To make it understandable, a program called a compiler is used. Each programming language has its own compiler.

In essence, a compiler translates your source file into code that is understandable by the system, and when done, places this code (called object code) into a file called a User Subprogram Library file. This file (commonly referred to as a USL file) is not executable in itself; it must be prepared, using the segmenter.

Preparation is similar to link-editing or binding on other systems; it is the process in which the object code is put into nearly-executable code. By nearly-executable, we mean that almost all references to external material used by the program (data files, for example) are resolved. There may be some references that cannot be resolved until the program is actually executed. The code that results from preparation is stored in a program file.

In both of these stages of program development (compilation and preparation), you have the option of specifying the name of a file to be used in storing the resulting code. If you don't specify a file name, two special system files, $NEWPASS and $OLDPASS, are created and used.

For example, let's assume that you have a source file that you have created by using the EDIT/3000 subsystem (Section 2), and that you wish to compile and prepare it. Also, assume that you do not specify a file to hold the resulting code.

When compilation begins, the compiler issues instructions to the MPE file system to create a file named $NEWPASS.

As compilation takes place, the compiler then places the translated code (the object code) into this new file. When compilation is complete, $NEWPASS is closed and renamed $OLDPASS. Thus, $OLDPASS is the USL file.

When you prepare the USL file ($OLDPASS), the prepared code is placed in $NEWPASS, just as the object code was placed in $NEWPASS in the compilation stage. (Remember that the previous file named $NEWPASS was renamed $OLDPASS.)

When preparation is complete, $NEWPASS is closed, the USL file ($OLDPASS) is purged, and $NEWPASS is then renamed $OLDPASS. Thus, $OLDPASS is now the program file, and can be executed.

$OLDPASS is a session-temporary file. By session-temporary, we mean that it is purged from the system when you end your session. It can be made permanent, however, by issuing a SAVE command. For example, the command:

```
:SAVE $OLDPASS,MYSAVE
```

renames the file called $OLDPASS, giving it the name MYSAVE. MYSAVE is a permanent file. Note that by permanent, we mean that MYSAVE remains in the system until you issue a PURGE command, which deletes the file. This is true no matter how many times you end or begin a session.

program control commands

Now, let's take a look at the specific commands that control the processing of a source file. To do so, we assume that you have created a source file called MYPROSRC.

Note: BASIC programs can be written and executed by using the BASIC interpreter as well as the BASIC compiler. In the following discussion, we are considering BASIC from the compiler point of view. For information on the BASIC interpreter, see Section 6.

COMPILATION

With the exception of BASIC, the command used to compile a source file is simply the name of the language that the program is written in, followed by the name of the source file (and optionally, by a second parameter that is the name of the file that you wish to have the object code kept in). For BASIC programs, the name of the command is BASICCOMP.

To compile the source file MYPROSRC, you issue one of the following commands:

```
:BASICOMP MYPROSRC
     or                          (if your source file is written in BASIC)
:BASICOMP MYPROSRC,MYPROUSL


:COBOL MYPROSRC
     or                          (if your source file is written in COBOL)
:COBOL MYPROSRC,MYPROUSL


:FORTRAN MYPROSRC
     or                          (If your source file is written in FORTRAN)
:FORTRAN MYPROSRC,MYPROUSL


:RPG MYPROSRC
     or                          (if your source file is written in RPG)
:RPG MYPROSRC,MYPROUSL


:SPL MYPROSRC
     or                          (if your source file is written in SPL)
:SPL MYPROSRC,MYPROUSL
```

In the first form of the commands above, the source file is compiled by the compiler for that particular language, and the object code is in $OLDPASS after compilation is complete.

In the second form, neither $NEWPASS nor $OLDPASS are used, and the object code is written into the permanent file named MYPROUSL. This is the file name of your choosing that you supply as a second parameter in the command. Note that if the file does not already exist when you specify it, the compiler creates it for you.

PREPARATION

To continue our discussion, let's assume that we used the first form of one of the above commands. Thus, the object code is in $OLDPASS.

Note that you must always specify the name of the file that is to receive the prepared code. Thus, you can either specify $NEWPASS (in which case, the prepared code is in $OLDPASS after preparation is complete, and the object code has been purged), or you can use a different file name of your choosing, in which case, the prepared code is in a file of that name when preparation is complete, and the object code is still in $OLDPASS.

To prepare the object code, you issue the command:

```
:PREP $OLDPASS,$NEWPASS

:PREP $OLDPASS,MYPROPRG
```

The first form of the prepare command prepares the object code from $OLDPASS into $NEWPASS. When preparation is complete, $NEWPASS is closed, $OLDPASS (the USL file) is purged, and $NEWPASS is renamed $OLDPASS. The USL file ($OLDPASS) was deleted because each file within an account must have a unique name, and since you specified $NEWPASS as the receiving file, $OLDPASS had to be used by the system when it renamed $NEWPASS. If you had used a SAVE command prior to issuing the PREP command (thereby renaming $OLDPASS), the USL file would not have been deleted.

The second form of the prepare command (:PREP $OLDPASS,-MYPROPRG) prepares the object code from $OLDPASS, and, without using $NEWPASS, places the prepared code into the named file, MYPROPRG. Thus, $OLDPASS still contains the object code.

Note that if you specify a file name for your program file, and it does not already exist, the segmenter creates a session-temporary file for you. Since this is not a permanent file, you may want to issue a SAVE command to make it so. To do so, you enter the SAVE command in the following form:

```
:SAVE MYPROPRG
```

Note that a second parameter is not needed in this form of the SAVE command. The only time that you need a second parameter is when the first is $OLDPASS (this is so because $OLDPASS cannot be made permanent. Otherwise, the system would have no file to rename $NEWPASS).

EXECUTION

When you execute a program, all external references are completely resolved (if not, the program terminates in an error condition), and the program is executed. To execute the program file, you issue the RUN command:

```
:RUN $OLDPASS

:RUN MYPROPRG
```

In the first form of the RUN command, $OLDPASS is the name of the program file; in the second, MYPROPRG is the name of the program file. Note that after execution, the program files are unchanged, and can be used again (but don't forget that $OLDPASS is session-temporary).

COMPILATION AND PREPARATION AS A SINGLE STEP

Each of the commands in this group allows you to compile and prepare a source file in one step. Thus, to compile and prepare MYPROSRC in one step, you issue one of the following commands:

```
:BASICPREP MYPROSRC
          or
:BASICPREP MYPROSRC,MYPROPRG
```

(if your source file is written in BASIC)

```
:COBOLPREP MYPROSRC
          or
:COBOLPREP MYPROSRC,MYPROPRG
```

(If your source file is written in COBOL)

```
:FORTPREP MYPROSRC
          or
:FORTPREP MYPROSRC,MYPROPRG
```

(if your source file is written in FORTRAN)

```
:RPGPREP MYPROSRC
          or
:RPGPREP MYPROSRC,MYPROPRG
```

(if your source file is written in RPG)

```
:SPLPREP MYPROSRC
          or
:SPLPREP MYPROSRC,MYPROPRG
```

(if your source file is written in SPL)

In the first form of each of the above commands, the source file is first compiled, using $NEWPASS and $OLDPASS. Next, $OLDPASS is used as the USL file, with preparation taking place as if you had issued the PREP command in the form:

```
:PREP $OLDPASS,$NEWPASS
```

That is, at the end of preparation, $OLDPASS is the program file.

In the second form, compilation proceeds as it does in the first form, so that the object code is in $OLDPASS after compilation is complete. At the preparation stage, though, $NEWPASS is not used, and the program file is written into MYPROPRG. Hence, $OLDPASS still contains the object code after the command has executed.

PREPARATION AND EXECUTION IN ONE STEP

A USL file can be prepared and executed in one step. This is done using the PREPRUN command. Note that there is no parameter here for naming the program file. Hence, the results of the preparation stage will always be in $OLDPASS.

To use this command, you issue it in the form:

```
:PREPRUN  $OLDPASS
```

```
:PREPRUN  MYPROUSL
```

You should use the first form if your object code is in $OLDPASS. Because $OLDPASS is the USL, and since $NEWPASS is used to receive the prepared code, $OLDPASS becomes the program file when preparation is complete. Hence, the USL file is no longer available.

You should use the second form if your object code is in, for example, the user-specified file MYPROUSL. Since the object code is not in $OLDPASS, it is not deleted with the completion of the preparation stage. The program file is written into $OLDPASS.

In either case, you can, of course, use a SAVE command to make a permanent copy of your program file.

COMPILATION, PREPARATION, AND EXECUTION IN ONE STEP

This group of commands provides the simplest method of program development, since it performs all of the three functions in one command. It does have some disadvantages over the other forms of commands, in that $OLDPASS is used to hold both the object code and the program file, which makes it impossible for you to save the object code. Also, if there are routines that you wish to add to your program (a subject that we don't cover here – see the compiler reference manual of the language that you're interested in for details), you cannot do so using this form of command.

To compile, prepare, and execute the source file MYPROSRC, you issue one of the following commands:

```
:BASICGO MYPROSRC    (if your source file is written in BASIC)

:COBOLGO MYPROSRC    (if your source file is written in COBOL)

:FORTGO MYPROSRC     (if your source file is written in FORTRAN)

:RPGGO MYPROSRC      (if your source file is written in RPG)

:SPLGO MYPROSRC      (if your source file is written in SPL)
```

When one of these commands executes, the compilation and preparation stages use $NEWPASS and $OLDPASS. After execution, you can issue a SAVE command to keep a permanent copy of your program file.

examples

Below are four examples of the commands that you might use to compile, prepare, and execute a source file. Each example is representative of one of the four combinations available for your use in program development.

Assume that MYPROSRC is the source file in each of the examples, and that it is a COBOL program.

EXAMPLE A:

```
:COBOLGO MYPROSRC
```

(compiles, prepares, and executes the source file MYPROSRC)

```
:SAVE $OLDPASS,MYPROPRG
```

(saves the program file in a permanent file named MYPROPRG)

EXAMPLE B:

```
:COBOL MYPROSRC
```

(compiles the source file, MYPROSRC)

```
:SAVE $OLDPASS,MYPROUSL
```

(saves the USL generated by the preceding command in a file named MYPROUSL)

```
:PREPRUN MYPROUSL
```

(prepares and executes the object code, MYPROSRC; places the program file into $OLDPASS)

```
:SAVE $OLDPASS,MYPROPRG
```

(saves the program file generated by the preceding command in a file named MYPROPRG)

Note: The first use of the SAVE command in this example is not really required. You could more easily use the USL file name, MYPROUSL, as a parameter in the COBOL command. However, it might be beneficial not to do so, in case there is some error in the compiled code. If there were an error, and you specified the name in the COBOL command, you would have to purge the file before using the same name in a later compilation command.

EXAMPLE C:

```
:COBOLPREP MYPROSRC,MYPROPRG
```

(compiles and prepares the source file MYPROSRC, placing the prepared code (program file) into MYPROPRG. Note that MYPROPRG is session temporary)

```
:RUN MYPROPRG
```

(executes the program file MYPROPRG)

```
:SAVE MYPROPRG
```

(makes MYPROPRG a permanent file)

EXAMPLE D:

```
:COBOL MYPROSRC,MYPROUSL
```

(compiles MYPROSRC into a USL file named MYPROUSL)

```
:PREP MYPROUSL,MYPROPRG
```

(prepares the object code in MYPROUSL into the session-temporary file, MYPROPRG)

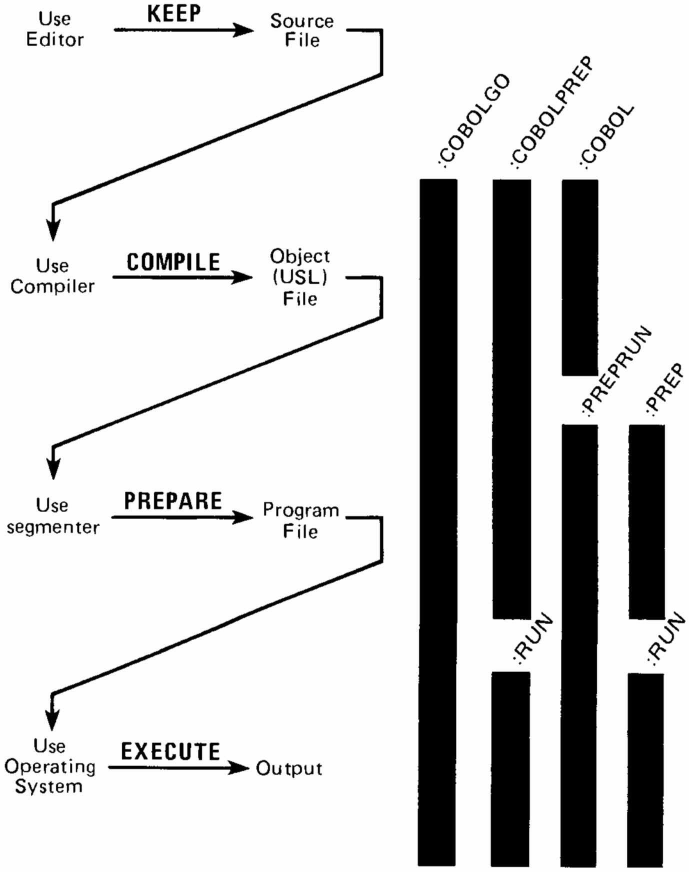
```
:RUN MYPROPRG
```

(executes the program file, MYPROPRG)

```
:SAVE MYPROPRG
```

(makes MYPROPRG a permanent file)

The graph below uses COBOL to illustrate the discussion that we presented in this section.



to summarize...

There are six groups of program control commands available for your use. These six groups can be used in four different ways, enabling you to perform the stages of program control exactly as you choose.

The six groups allow you to:

- Compile a source file in one step;
- Prepare the object code in one step;
- Execute the prepared code in one step;
- Compile and prepare a source file in one step;
- Prepare and execute the object code in one step;
- Compile, prepare, and execute a source file in one step.

All of the commands available for each group are listed on page 3-3, in the order that they appear above.

The four different ways in which the commands can be applied are:

- Compile, prepare, and execute in separate steps;
- Compile in one step, and then prepare and execute in the next;
- Compile and prepare in one step, and execute in the next;
- Compile, prepare, and execute in one step.

finally...

You should know that this section describes only the basic use of the program control commands. There are many other parameters to all of the commands discussed, and there are as many reasons for using a particular configuration of commands as there are programmers. As you become more experienced with the HP 3000, and your own programming language, you will, no doubt, learn to choose the proper commands for a particular program.

section 4
using FORTRAN/3000

summary of commands introduced in this section

ACCEPT P,Q,R	prompts with "?" and accepts three free-field data items from terminal keyboard.
$CONTROL MAP	lists program's symbol names after compilation.
$CONTROL NOSOURCE	suppresses source listing during compilation.
$CONTROL USLINIT	clears object file before compilation; if used, should be first statement in program.
DISPLAY "ANSWER:",S	writes strings and data items to terminal display unit (without format statement).
FORMAT("ANSWER:",F12.3)	format statement for outputting a string and fixed point formatted data item.
FORMAT("ENTER"/)	format statement without data specification requires slash for carriage-return/line-feed; first character in quotes should be blank if not used for carriage control.
READ (5,100)A	reads A from terminal using format statement 100.
READ (8,100,END=400)A	transfers control to statement label 400 if end-of-file encountered on reading file 8.
READ (2@5,100)C	reads record 5 in file 2; record number may be an expression which is truncated to an integer.
WRITE (6,200)A	writes A to terminal using format statement 200.
:BUILD DATAFL;DISC=100	builds a disc file of 100 records called DATAFL (external to FORTRAN).
:FILE LP;DEV=LP	creates a file reference for the line printer.
FILE FTN02=DATAFL,OLD	equates FORTRAN file 2 to old file which was constructed by BUILD command.

summary of commands

:FILE FTN02;DEV=LP	equates FORTRAN file 2 to the line printer.
:FORTGO TRY1,*LP	compiles, prepares, and executes source file TRY1. Prints source listing at line printer by back referencing LP.
:FORTPREP S1	compiles and prepares source file S1; resulting program file is system file called $OLDPASS.
:FORTPREP S1,P2	compiles and prepares source file S1; resulting program file is session-temporary file named P2.
:FORTRAN S1	compiles source file S1; stores resulting object code in $OLDPASS.
:PREP C2,P3	prepares object file C2 into session-temporary file P3.
:PREP $OLDPASS,P4	prepares object file from most recent compilation into session-temporary file P4.
:PREPRUN C1	prepares and executes object file C1; assumes $OLDPASS from compilation was saved in C1.
:PREPRUN $OLDPASS	prepares and executes object file from most recent compilation (where object file was not saved into a named file).
:RUN P2	executes program file P2.
:SAVE $OLDPASS,NEW	saves most recent compilation or preparation (where no permanent file was specified) in new permanent file NEW.
:SAVE P4	saves file P4 in new permanent file.

using FORTRAN/3000

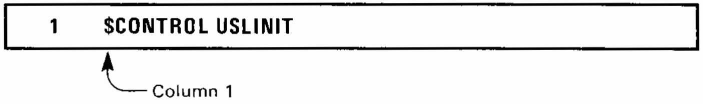
FORTRAN/3000 is a compiler language which accepts source statements at the full ANSI FORTRAN IV level plus some extensions found only in FORTRAN/3000. We assume that you are familiar with FORTRAN IV.

The usual sequence of operations is to write a source program using EDIT/3000, compile this program using the FORTRAN/3000 compiler, and execute it using MPE/3000 Operating System commands.

things to remember

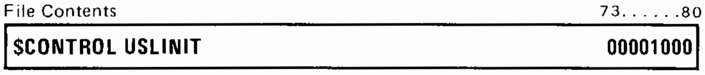
Column 1. When entering a source program using the Editor, the Editor always reserves the first ten character positions for the line number. After printing out the line number, the carriage or cursor stops at what is physically character position 11 on the terminal. However, we will always refer to this as the column 1 position. It is the actual beginning point of the record. The Editor actually stores the line number at the *end* of the record in disc files, to make the disc files look like card images.

Terminal Display



```
    1     $CONTROL USLINIT
```

File Contents



```
$CONTROL USLINIT                                            00001000
```

Label Field. Columns 1 through 5 are reserved for statement labels, which can have up to five digits (1 to 99999). Column 6 is reserved for a continuation indicator. You must space over these six positions for statements which do not have labels. The only other legal uses of this field are for comments and compiler commands, in which cases the entire line is available if you type a C or $ in column 1.

Initialization. The first statement of a FORTRAN program should be $CONTROL USLINIT. This ensures that the object (USL) file receiving the output from the compiler is initialized to an empty condition before compilation starts. (An exception is if you want to compile several program units to the same object file at various times.) The statement starts in column 1.

Unit Numbers. The allocation of unit numbers is as follows: for programs executed from a terminal, unit 5 is the keyboard and unit 6 is the terminal's output device (printer or display). Therefore your READ/WRITE will cause an error if you request a READ on unit 6 or a WRITE on unit 5.

writing a source program

You write a source program using the Editor as you did in Section 2. That is, you first get into the Editor subsystem by typing EDITOR in response to a colon prompt. Then you type ADD and begin typing your program in response to line number prompts. Try this:

```
:

HP32201A.7.01 EDIT/3000 TUE,  MAR 3, 1978 11:03 AM
(C) HEWLETT-PACKARD CO. 1976
/
     1       [illegible]
     2       [illegible]
     3              [illegible]
     4              [illegible]
     5              [illegible]
     6              WRITE(6,[illegible]
     7       [illegible]     [illegible]
     8       [illegible]     [illegible]
     9              [illegible]
    1Ø              END
    11       ...
/
```

(We expect this program to calculate and print out the sum of the sine and cosine of 1, then of 2, then of 3; a fixed-point format is to be used for printing, with a field width of five digits, rounded to two fractional digits.)

The program, now in your work area, may be edited if necessary. When you are satisfied with it, save the source program in a disc file with the KEEP command.

```
/KEEP TRY1
/
```

Now terminate use of the Editor with an END command.

```
/END

 END OF SUBSYSTEM
:
```

to compile and execute

Just type FORTGO followed by the file name (TRY1) of your source program.

Upon pressing the RETURN key, you will see the following things happen, as illustrated in the sample printout on this page:

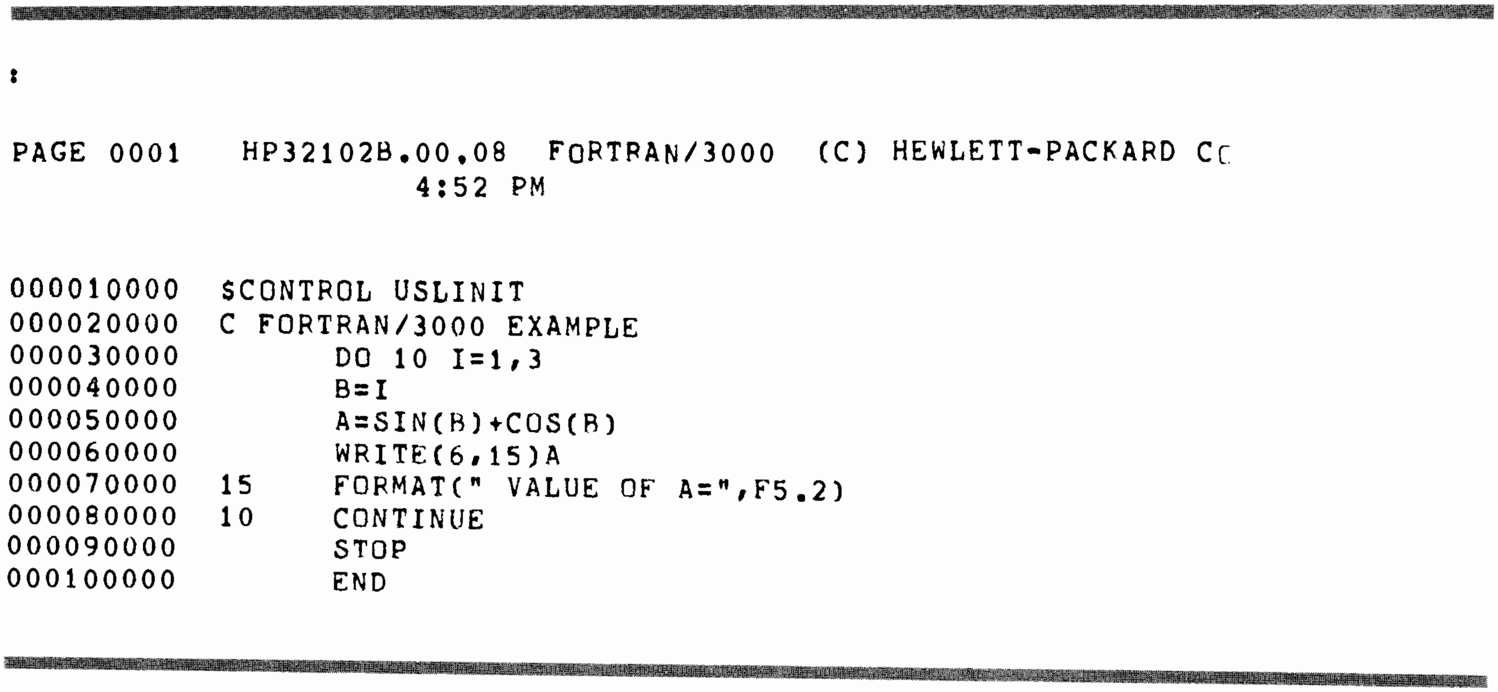
a. A four-digit page number is printed out, followed by the revision identification number of the FORTRAN/3000 compiler. Page numbering is inserted at intervals appropriate for standard 11-inch page length.

b. Each line of the source program is printed out as it is compiled. Also, each line is preceded by the line number assigned to it by the Editor, except that the number is printed in a special eight-digit format. A decimal point is assumed between the third and fourth digits from the right, and leading and trailing zeros are filled in to make eight digits.

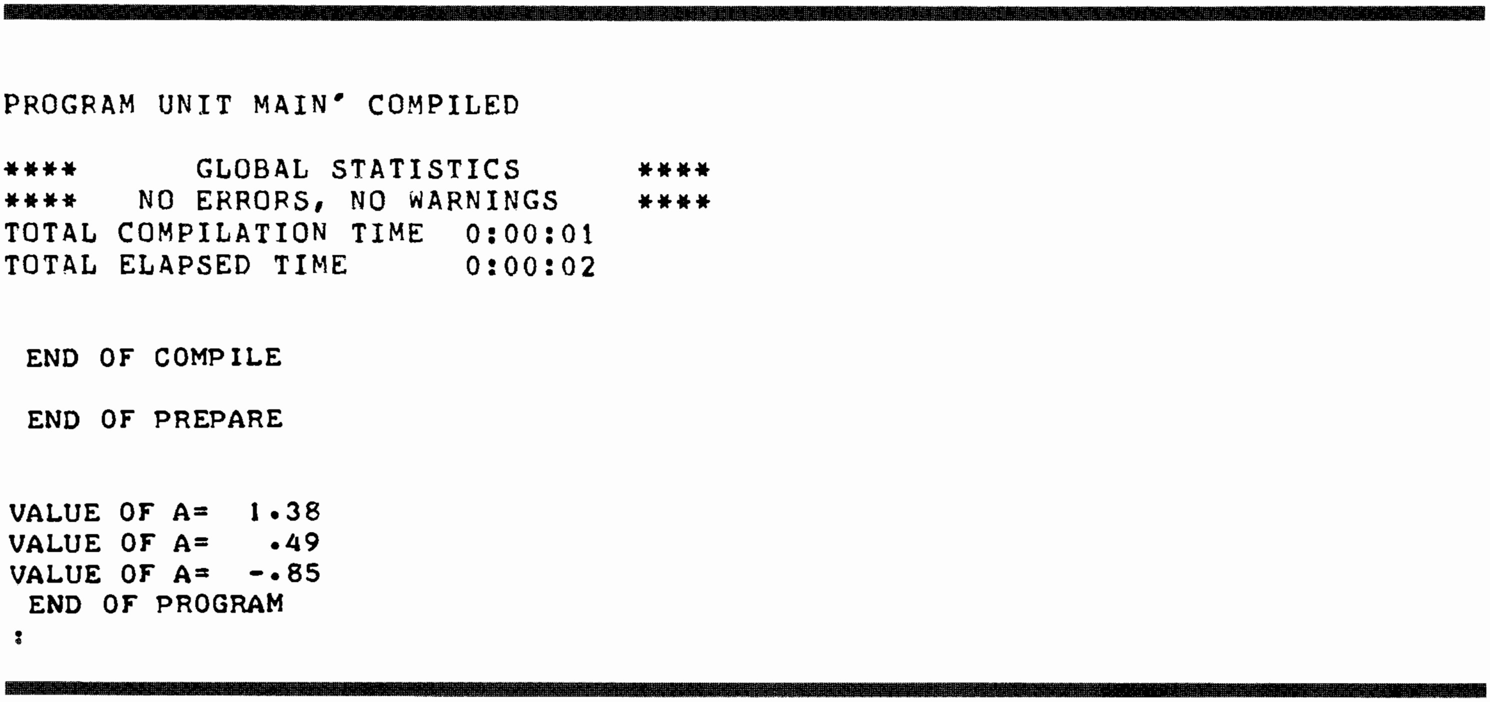
c. Next, several statistics about the compilation are printed. The message PROGRAM UNIT MAIN' COMPILED means the compilation was successful; if serious errors in your source program prevented successful compilation, you would get the message PROGRAM UNIT FLUSHED.

d. The messages END OF COMPILE and END OF PREPARE are next printed. These mean, respectively, that your use of the compiler and the segmenter are successfully terminated. (We will be discussing the role of the segmenter shortly.) If you were unsuccessful, you would get an error message which reads, "PROGRAM TERMINATED IN AN ERROR STATE. (CIERR976)."

e. Next you see the results of executing the sample program. Three values for A are printed out, followed by the message END OF PROGRAM.



```
:

PAGE 0001   HP32102B.00.08  FORTRAN/3000  (C) HEWLETT-PACKARD Cc
                     4:52 PM


000010000  $CONTROL USLINIT
000020000  C FORTRAN/3000 EXAMPLE
000030000        DO 10 I=1,3
000040000        B=I
000050000        A=SIN(B)+COS(B)
000060000        WRITE(6,15)A
000070000  15    FORMAT(" VALUE OF A=",F5.2)
000080000  10    CONTINUE
000090000        STOP
000100000        END
```



```
PROGRAM UNIT MAIN' COMPILED

****       GLOBAL STATISTICS       ****
****    NO ERRORS, NO WARNINGS     ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:02


 END OF COMPILE

 END OF PREPARE


VALUE OF A=  1.38
VALUE OF A=   .49
VALUE OF A=  -.85
 END OF PROGRAM
:
```

to save the program file

After performing the FORTGO, the runnable version of the program (or "prepared" program) will be in a system temporary file called $OLDPASS. This can be changed into a permanent file belonging to you by entering:

```
:SAVE $OLDPASS,NEWNAME
:
```

The file name of the permanent file is an arbitrary name, NEWNAME in this example. Like all system file names, it must begin with an alphabetic letter, must be no longer than eight characters, and must not already exist.

compilation, preparation, and execution as separate steps

Using :FORTGO, as you did, simplifies matters for you. All the necessary file manipulations are accomplished automatically with no intervention required from you.

However, there will come times when you may wish to exercise some control over the various phases of program management. For example, :FORTGO does not allow use of Group or Public library routines, and the compiled object code can neither be accessed nor saved (although the "prepared" program can be). Depending on the type of control you want, you can break down the sequence of program management into any combination of steps possible. Six operating system commands give you that control. (See Section 3).

They are:

:FORTGO
:FORTPREP
:FORTRAN
:PREPRUN
:PREP
:RUN

:FORTGO compiles, prepares and executes; :FORTPREP compiles and prepares; :FORTRAN compiles only; :PREPRUN prepares and executes; and :RUN executes only.

The important characteristics of these commands are listed in the table on this page. We will not, in this manual, demonstrate all of these characteristics; however, it is useful to be aware of what you can and cannot control. Check the operating system reference manual for the methods of library searches and complete command formats. (And consult your system manager to determine what library routines are available to you.)

But just to get the feel of using these commands, we will now try them all. It's very easy and takes only a few minutes.

First, though, a word about temporary files. You have already used one: $OLDPASS. This is a default file used to receive output code in the compile and prepare steps in the event you do not (or are not permitted to) specify a permanent file. (You will be shown how to specify permanent files in the examples which follow.) If you want to save $OLDPASS in a permanent file, you must do so before another compile or prepare, as this would destroy the existing contents. Job-temporary files, as generated for :FORTPREP and :PREP, remain in existence for the duration of your session only, but also can be saved if desired.

USING :FORTGO

1. Compiles, prepares, and executes.
2. Object file inaccessible; disappears.
3. Program file may be saved (:SAVE).
4. Programs may *not* reference Group or Public library routines, or Relocatable Library.

USING :FORTPREP

1. Compiles and prepares.
2. Object file may be saved (add a parameter).
3. Resulting program file is job-temporary.
4. Program may reference library routines available to :RUN command at run time.

USING :FORTRAN

1. Compiles only.
2. Object file may be saved (:SAVE).
3. Object file must be prepared into program file before execution can occur.

USING :PREPRUN

1. Prepares program file (from object file) and executes same.
2. Program file may be saved (:SAVE).
3. Program may reference routines in Group, Public, or System Library, or Relocatable Library (LIB= or RL=).

USING :PREP

1. Prepares program file (from object file) only.
2. Resulting program file is job-temporary.
3. Program may reference Relocatable Library routine (RL=) plus, at run time, those libraries available to the :RUN command.

USING :RUN

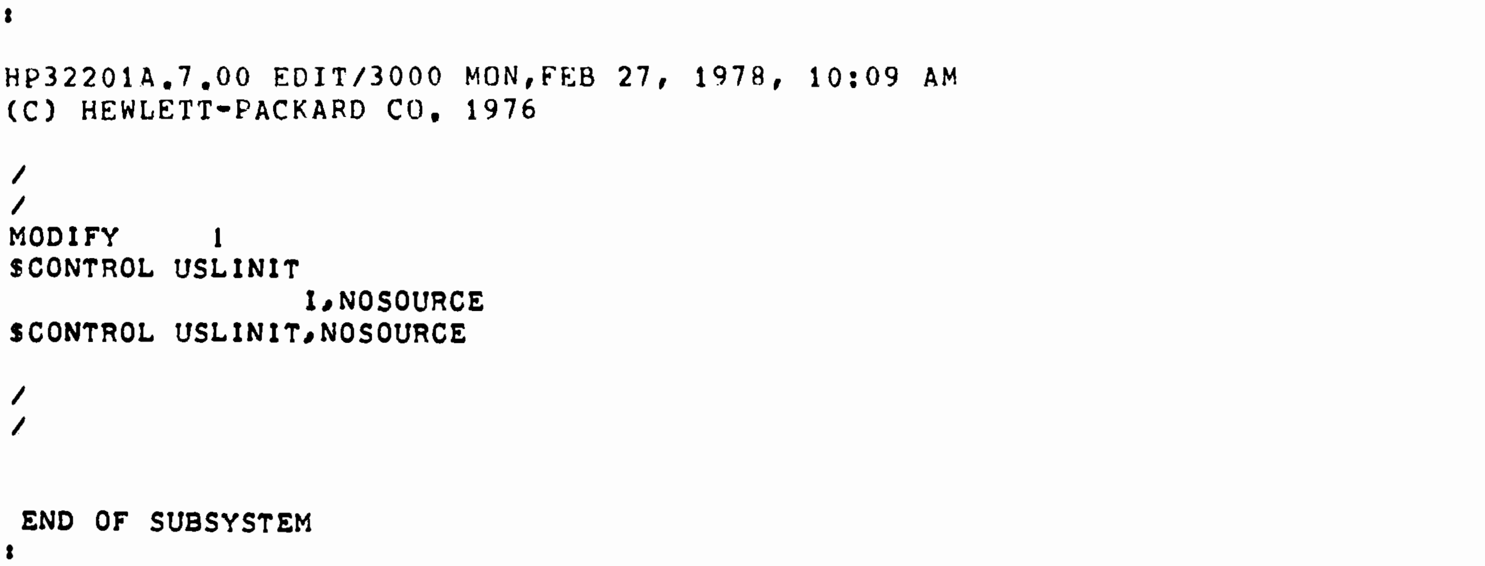
1. Executes prepared program files only.
2. Program may reference routines in Group, Public, or System Library (LIB=).
3. Program may *not* reference Relocatable Library routines (RL=) unless resolved by :PREP.

COMPILE-PREPARE-EXECUTE

If you did not do the operations on page 4-7, do so now. You will need the file TRY1 for the following operations.

COMPILE-PREPARE, THEN EXECUTE

Since the compiler normally prints a source listing on each compilation, we should take advantage of a compiler option that suppresses this listing. (Otherwise you will get very tired of seeing it.) After logging on, respond to the prompts as follows:



```
:

HP32201A.7.00 EDIT/3000 MON,FEB 27, 1978, 10:09 AM
(C) HEWLETT-PACKARD CO. 1976

/
/
MODIFY     1
$CONTROL USLINIT
                I,NOSOURCE
$CONTROL USLINIT,NOSOURCE

/
/


 END OF SUBSYSTEM
:
```

The modified source file is now the disc file S1. (The original is still in TRY1.) Line 1 of the new source file is $CONTROL USLINIT,NOSOURCE. The NOSOURCE parameter states that we do not want a source listing on compilation.

Now proceed. Compile and prepare with a :FORTPREP S1 command.

```
:

PAGE 0001    HP32102B.00.08 FORTRAN/3000  (C) HEWLETT-PACKARD CO

00001000  $CONTROL USLINIT,NOSOURCE

****       GLOBAL STATISTICS       ****
****    NO ERRORS, NO WARNINGS     ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:03

PROGRAM UNIT MAIN' COMPILED

 END OF COMPILE

 END OF PREPARE
:
```

Since we didn't specify a permanent program file, the program is now in the temporary file $OLDPASS. Run $OLDPASS as follows:

```
:


VALUE OF A=  1.38
VALUE OF A=   .49
VALUE OF A=  -.85

 END OF PROGRAM
:
```

You can save the program file if you like.

```
:
:
```

That puts the program in the permanent file P1; you can then RUN P1 any time.

COMPILE, THEN PREPARE-EXECUTE

Compile by typing FORTRAN S1.

```
:

PAGE 0001   HP32102B,00,08 (C) HEWLETT-PACKARD CO. 1976

ØØØØ1ØØØ  $CONTROL USLINIT,NOSOURCE


PROGRAM UNIT MAIN' COMPILED

****       GLOBAL STATISTICS       ****
****    NO ERRORS, NO WARNINGS     ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:03

 END OF COMPILE
:
```

The object code is now in $OLDPASS. You could, if you wanted to, save the object code in a permanent file at this time (e.g., :SAVE $OLDPASS,C1). But let's say you do not need a permanent copy of the object code; you can then prepare-execute $OLDPASS directly.

```
:

 END OF PREPARE

VALUE OF A=  1.38
VALUE OF A=   .49
VALUE OF A=  -.85
 END OF PROGRAM
:
```

(If you wanted a permanent copy of the program file, you could save $OLDPASS now.)

COMPILE, THEN PREPARE, THEN EXECUTE

Compile by typing FORTRAN S1.

```
:

PAGE 0001    HP32102B.00.08 FORTRAN/3000  (C) HEWLETT-PACKARD CO, 1976

00001000  $CONTROL USLINIT,NOSOURCE


****      GLOBAL STATISTICS      ****
****   NO ERRORS, NO WARNINGS   ****
TOTAL COMPILATION TIME 0:00:01
TOTAL ELAPSED TIME     0:00:02

 END OF COMPILE
:
```

If you do not need a permanent copy of the object code, you can prepare $OLDPASS directly. (Optionally, you could save the object code now with a :SAVE $OLDPASS,C2.) Use :PREP as follows.

```
:PREP $OLDPASS,P4

 END OF PREPARE
:
```

Note: Unlike :FORTPREP and :PREPRUN, :PREP requires an output file to be specified; there is no default. But remember — the output file so specified is job-temporary. That is, it disappears when you log off. If you want a permanent copy, save the program file (:SAVE P4) before you log off.

Then, at any time during the session, you can execute the program file P4.

```
:RUN P4


VALUE OF A=  1.38
VALUE OF A=   .49
VALUE OF A=  -.85

 END OF PROGRAM
:
```

obtaining hard-copies of your source file

As a FORTRAN programmer, you are well aware that new source files often contain one or more errors.

It is often more convenient to be able to study a printed version of a program while you search for errors than to study the source file at the terminal. This is particularly true when your source file is very long.

There are essentially two ways to obtain a printed copy of your source file. The first, which uses the Editor, has already been discussed on page 2-16. The other way is to use the listfile parameter in each of the operating system's FORTRAN commands (FORTGO, FORTPREP, and FORTRAN). In either case, you must first use a FILE command to reference the line printer.

Since we used the NOSOURCE option in the $CONTROL statement of the S1 source file, we will use TRY1, the original source file, in the MPE commands that compile, prepare, and execute FORTRAN source files.

Enter the commands exactly as shown below, being sure to insert the needed commas. Afterwards, you may pick up your source listings (there will be three of them) from the line printer.

```
:

:


****       GLOBAL STATISTICS         ****
****    NO ERRORS,    NO WARNINGS    ****
TOTAL COMPILATION TIME   0:00:01
TOTAL ELAPSED TIME       0:00:02
:FILE LP;DEV=LP
```

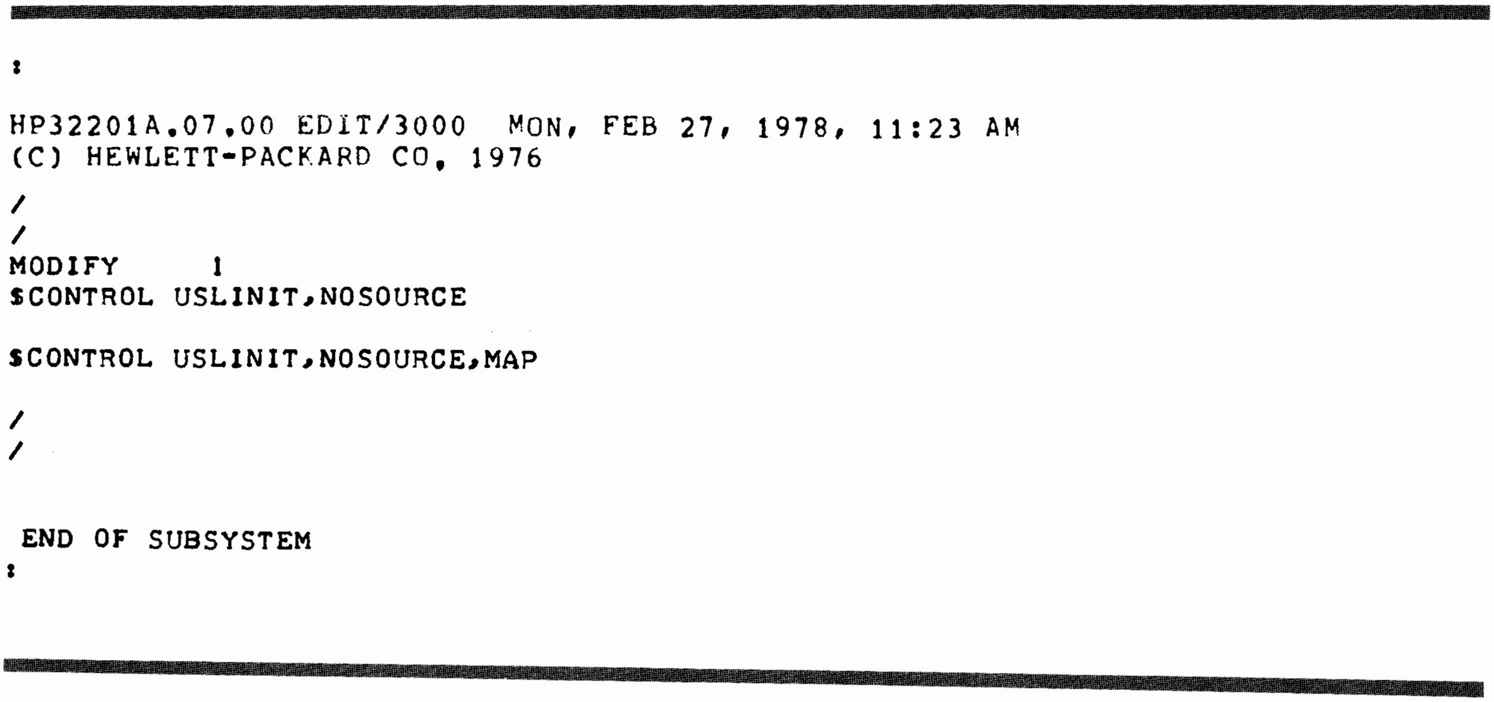
```
END OF COMPILE

END OF PREPARE



VALUE OF A= 1.38
VALUE OF A=  .49
VALUE OF A= -.85

END OF PROGRAM
:


:


****    NO ERRORS,    NO WARNINGS    ****
TOTAL COMPILATION TIME   0:00:01
TOTAL ELAPSED TIME       0:00:09

END OF COMPILE

END OF PREPARE
:


:


****    NO ERRORS,    NO WARNINGS    ****
TOTAL COMPILATION TIME   0:00:01
TOTAL ELAPSED TIME       0:00:09

END OF COMPILE
:
```

For further information on how the listfile parameter may be used, and to see the complete list of possible parameters for each of the MPE FORTRAN commands, check the *MPE Commands Reference Manual.*

listing symbolic names

If you want an alphabetical listing of all symbolic names used in a program, you can get it at compile time by including a $CONTROL MAP statement in the program. This may be combined with other $CONTROL parameters like USLINIT and NOSOURCE.

To illustrate, let's get the Editor back and do the following.



```
:

HP32201A.07.00 EDIT/3000  MON, FEB 27, 1978, 11:23 AM
(C) HEWLETT-PACKARD CO, 1976
/
/
MODIFY     1
$CONTROL USLINIT,NOSOURCE

$CONTROL USLINIT,NOSOURCE,MAP

/
/


 END OF SUBSYSTEM
:
```

Now compile with :FORTRAN S2 and watch what happens.

```
PAGE 0001   HP32102B.00.08  FORTRAN/3000  (C) HEWLETT-PACKARD CO.

00001000  $CONTROL USLINIT,NOSOURCE,MAP

  SYMBOL MAP

NAME           TYPE       STRUCTURE   ADDRESS
A              REAL       SIMPLE VAR  Q+%4
B              REAL       SIMPLE VAR  Q+%2
COS            REAL       FUNCTION
I              INTEGER    SIMPLE VAR  Q+%1
SIN            REAL       FUNCTION


PROGRAM UNIT MAIN' COMPILED

****      GLOBAL STATISTICS      ****
****   NO ERRORS,   NO WARNINGS  ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:02

 END OF COMPILE
:
```

The symbol map shows that we used five symbol names in the program. Four of them are type real, one is type integer. Two are structured as functions, three as simple variables. (The "address" is a hardware location relative to a pointer called Q; ignore it for now.)

Incidentally, the $CONTROL MAP statment may be placed anywhere in the program since it does not take effect until after the program unit is compiled. Also, if you choose to have your source file printed out, and the MAP option is specified within it, the symbol map is printed with the source file; it is not displayed at your terminal.

summary

$CONTROL USLINIT	clears object file before compilation
$CONTROL NOSOURCE	suppresses source listing during compilation
$CONTROL MAP	lists program's symbol names after compilation
:FORTGO TRY1	compiles, prepares, executes source file TYR1
:SAVE $OLDPASS,NEWNAME	saves most recent compilation or preparation (where no permanent file was specified) in new permanent file NEWNAME
:SAVE P4,	saves file P4 in new permanent file
:FORTPREP S1	compiles and prepares source file S1; resulting program file is $OLDPASS
:FORTPREP S1,P2	compiles and prepares source file S1; resulting program file is job-temporary file P2
:RUN $OLDPASS	executes program file from most recent preparation (where no permanent file was specified)
:RUN P2	executes program file P2
:FORTRAN S1	compiles source file S1
:PREPRUN $OLDPASS	prepares and executes object file from most recent compilation (where object file was not saved)
:PREPRUN C1	prepares and executes object file C1; assumes $OLDPASS from compilation was saved in C1
:PREP $OLDPASS,P4	prepares object file from most recent compilation into job-temporary file P4
:PREP C2,P3	prepares object file C2 into job-temporary file P3

input/output with FORTRAN/3000

For the next few pages we will consider some of the most basic input/output operations. Specifically we will try:

a. simple terminal I/O

b. writing data into a disc file

c. reading data out of a disc file

Obviously, this does not cover all possible cases. Later you will want to do file-to-file, core-to-core, tape-to-file (and vice versa), card-to-file, and so on. However, these few operations will get you started. Once you are familiar with the basics, you can look up what you need in the FORTRAN/3000 reference manual.

Also, while you are learning to do these three I/O operations, we will throw some other useful concepts at you, such as:

a. using the ACCEPT and DISPLAY statements

b. using the END= and ERR= parameters

c. Using subroutines

d. equating FORTRAN and system files

e. building data files

f. directly accessing records in data files

TERMINAL I/O

The first I/O example will consist of writing, compiling, and running a program that accepts data items from the terminal, performs a computation, and displays the result on the terminal.

As mentioned previously, the terminal keyboard is logical unit number 5 and the terminal display mechanism is logical unit number 6. Thus, READ and WRITE statements for terminal I/O can look similar to the statements you are used to, such as

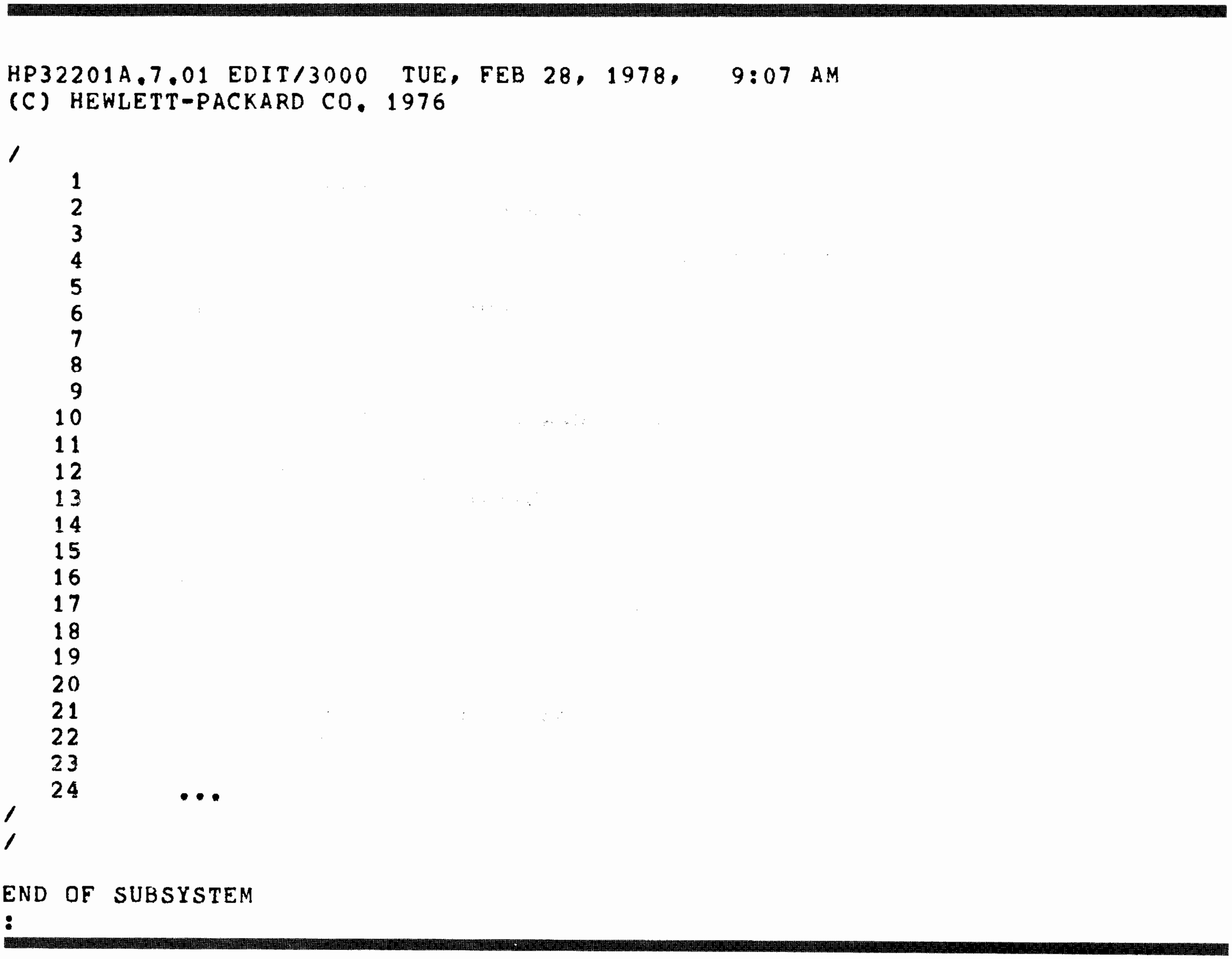
```
READ (5,100)A
WRITE (6,200)B
```

where 5 and 6 are the logical unit numbers, 100 and 200 are FORMAT specification references, and A and B are the data items for input and output.

To make terminal I/O more convenient, FORTRAN/3000 provides an ACCEPT and a DISPLAY statement. These are like READ and WRITE except that the terminal is understood to be the I/O device, making it unnecessary to specify a unit number. Also, ACCEPT prints out a "?" prompt and permits several data items to be input in free-field format on the same line, separated by commas. A complete list of differences is given in the following table.

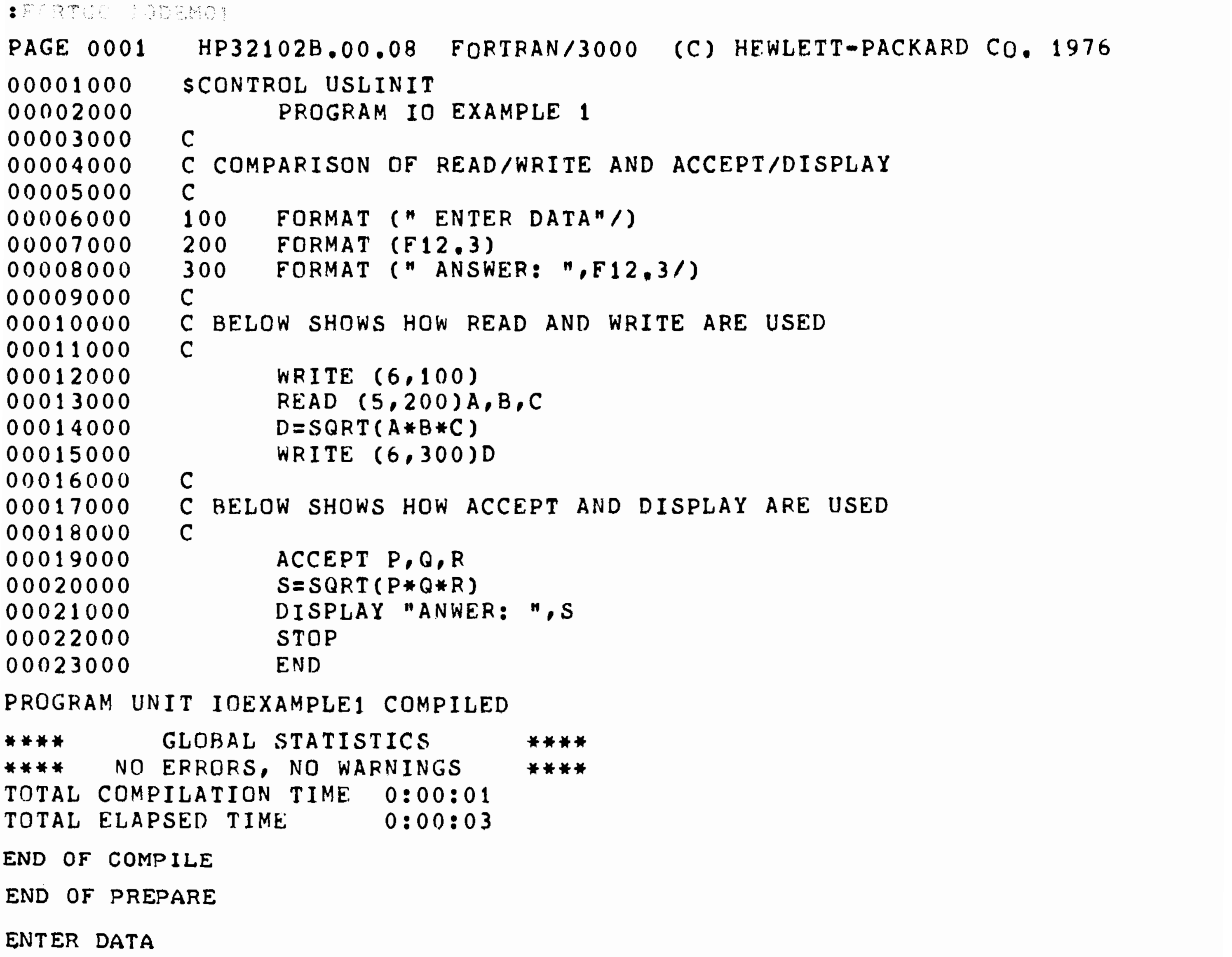
READ/WRITE	**ACCEPT/DISPLAY**
Requires unit number	Uses no unit number
Usually requires FORMAT statement	Uses no FORMAT statement
Permits one data item per line (unless multiple fields specified in FORMAT)	Permits any number of data items per line
No prompt for READ	? prompt for ACCEPT
Rounds to fit FORMAT specification	Rounds to 6 significant digits
Allows carriage control (in FORMAT statement)	No carriage control
Allows END=, ERR=	Does not allow END=, ERR=

Now try a sample program. After logging on, type EDITOR and enter the following program.



```
HP32201A.7.01 EDIT/3000  TUE, FEB 28, 1978,   9:07 AM
(C) HEWLETT-PACKARD CO. 1976

/
    1
    2
    3
    4
    5
    6
    7
    8
    9
   10
   11
   12
   13
   14
   15
   16
   17
   18
   19
   20
   21
   22
   23
   24        ...
/
/

END OF SUBSYSTEM
:
```

Then compile, prepare, and run by using :FORTGO. When the program stops after printing ENTER DATA, it is waiting for you to enter data to satisfy the READ statement in line 13. (The slash in the FORMAT statement of line 6 specifies a carriage-return and line-feed after printing out the ENTER DATA string.)



```
:FORTGO IODEMO1

PAGE 0001    HP32102B.00.08   FORTRAN/3000   (C) HEWLETT-PACKARD CO. 1976

00001000    $CONTROL USLINIT
00002000          PROGRAM IO EXAMPLE 1
00003000    C
00004000    C COMPARISON OF READ/WRITE AND ACCEPT/DISPLAY
00005000    C
00006000    100   FORMAT (" ENTER DATA"/)
00007000    200   FORMAT (F12.3)
00008000    300   FORMAT (" ANSWER: ",F12.3/)
00009000    C
00010000    C BELOW SHOWS HOW READ AND WRITE ARE USED
00011000    C
00012000          WRITE (6,100)
00013000          READ (5,200)A,B,C
00014000          D=SQRT(A*B*C)
00015000          WRITE (6,300)D
00016000    C
00017000    C BELOW SHOWS HOW ACCEPT AND DISPLAY ARE USED
00018000    C
00019000          ACCEPT P,Q,R
00020000          S=SQRT(P*Q*R)
00021000          DISPLAY "ANWER: ",S
00022000          STOP
00023000          END
PROGRAM UNIT IOEXAMPLE1 COMPILED

****       GLOBAL STATISTICS       ****
****    NO ERRORS, NO WARNINGS     ****
TOTAL COMPILATION TIME   0:00:01
TOTAL ELAPSED TIME       0:00:03

END OF COMPILE

END OF PREPARE

ENTER DATA
```

At this point, you must enter three items of data on three separate lines as shown below. After you do, the program will do its square-root computation and print out the result as specified by the WRITE statement in line 15.

```
ANSWER:        4922.488
```

When the program again stops after printing out a ? prompt, it is waiting for you to enter more data to satisfy the ACCEPT statement in line 19.

Now enter three more items of data on the same line, separated by commas and (just for legibility) one or more blanks. Again, computation occurs when you press RETURN, and the result is printed out according to the DISPLAY statement in line 21. And that's the end of the program.

```
?

ANSWER:    4922.49
 END OF PROGRAM
:
```

At this point it would be good to review the table of differences between READ/WRITE and ACCEPT/DISPLAY, and compare each point with the results obtained when you ran your program.

Some additional notes:

a. The character after the first quote mark in a FORMAT statement is interpreted as a carriage control character, as follows:

Blank	Single space
0	Double space
1	Page eject
+	Suppress Space

For the examples used here, leave this character blank.

b. A slash, for carriage-return/line-feed, may be positioned anywhere in a FORMAT statement. Multiple slashes may be used; e.g., /// or 3/ for three carriage-return/lines-feeds.

c. The PROGRAM statement (line 2) is optional. If not given, a default name of MAIN' is used internally. If you do use a PROGRAM statement, the name you assign may be up to 15 characters long, must start with an alphabetic character, and may contain numerals and embedded blanks (as shown in the example: IO EXAMPLE 1).

WRITING DATA INTO A FILE

FORTRAN logical unit numbers, in a particular run, can have values from 1 to 99. By convention, in session mode, logical unit numbers 5 and 6 are usually understood to mean the terminal. The remaining numbers, if used, are assigned by you to such files as tapes, discs, line printers, card punches, and so forth.

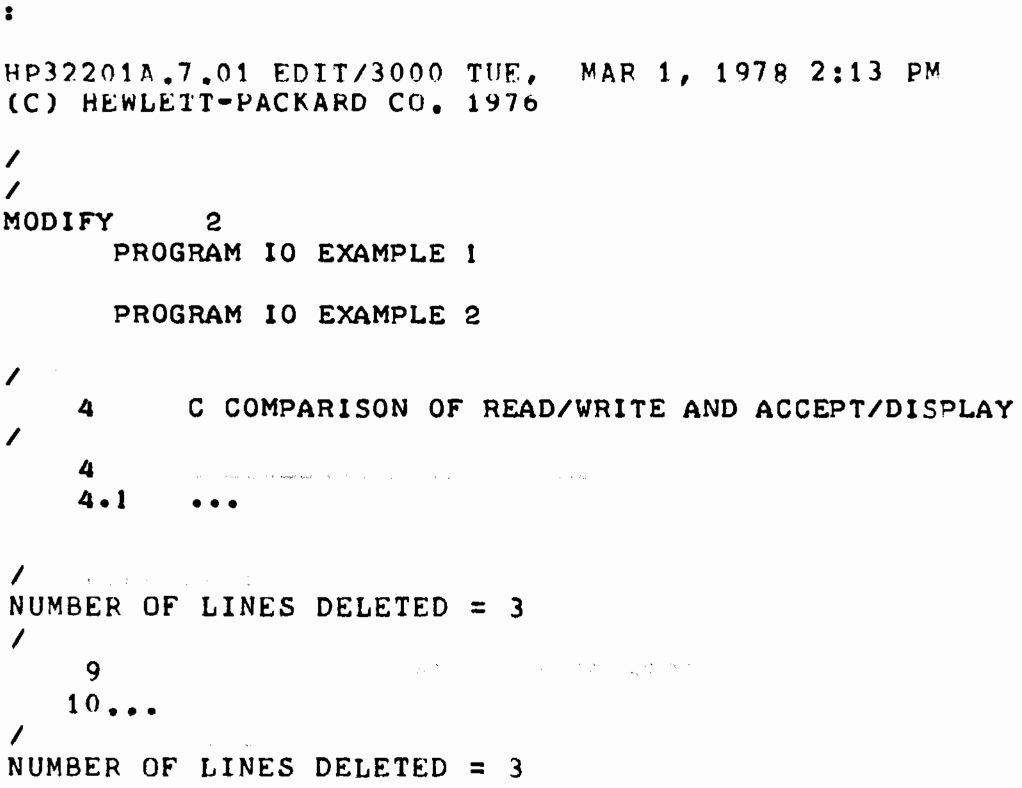
A WRITE statement referencing one of these numbers (if not otherwise previously defined by you) automatically opens a disc file of 1023 records. For example:

```
WRITE (8,100)A
```

opens FORTRAN FILE 8 as a disc file. In this case, the name, FTN08, by which your program knows the file (called the "formal file designator") is the same as the name by which the operating system knows your file (called the "actual file designator"). That is, the characters FTN are prefaced to the two-digit FORTRAN file number to form the file name, and in this form the file can be manipulated outside your program.

So, let's build a data file this way and see how it works. While we're at it, we'll throw in a subroutine to see how it compiles and executes.

Assuming you still have the source file IODEMO1 from the previous exercise, call the Editor and modify the file as follows.



```
:

HP32201A.7.01 EDIT/3000 TUE,  MAR 1, 1978 2:13 PM
(C) HEWLETT-PACKARD CO. 1976

/
/
MODIFY     2
      PROGRAM IO EXAMPLE 1

      PROGRAM IO EXAMPLE 2

/
    4      C COMPARISON OF READ/WRITE AND ACCEPT/DISPLAY
/
    4
    4.1    ...

/
NUMBER OF LINES DELETED = 3
/
    9
   10...
/
NUMBER OF LINES DELETED = 3
```

```
/ADD 16
   16                                   [illegible]
   17
   18...
/DELETE 19,20
NUMBER OF LINES DELETED = 2
/ADD 22
   22              [illegible]
   23
   24
   25
   26
   27
   28
   29
   30
   31
   32...
/
/
/

END OF SUBSYSTEM
:
```

These modifications accomplish the following:

Line 2
renames the program to IO EXAMPLE 2.

Line 4
changes the descriptive comment to BUILDING A DATA FILE.

Line 9
adds useful dialogue.

Line 16
creates file 2 and stores data elements A, B, C, and D into it.

Line 17
prints out the dialogue in line 9.

Line 22
(new) stores four more data elements into file 2; it also specifies a statement label (500) to transfer to in case end-of-file is encountered.

Note: An ERR= parameter is also available. This provides a statement label to transfer to in case of a file access failure; it does not catch format errors. It may be included in place of or following the END= parameter.

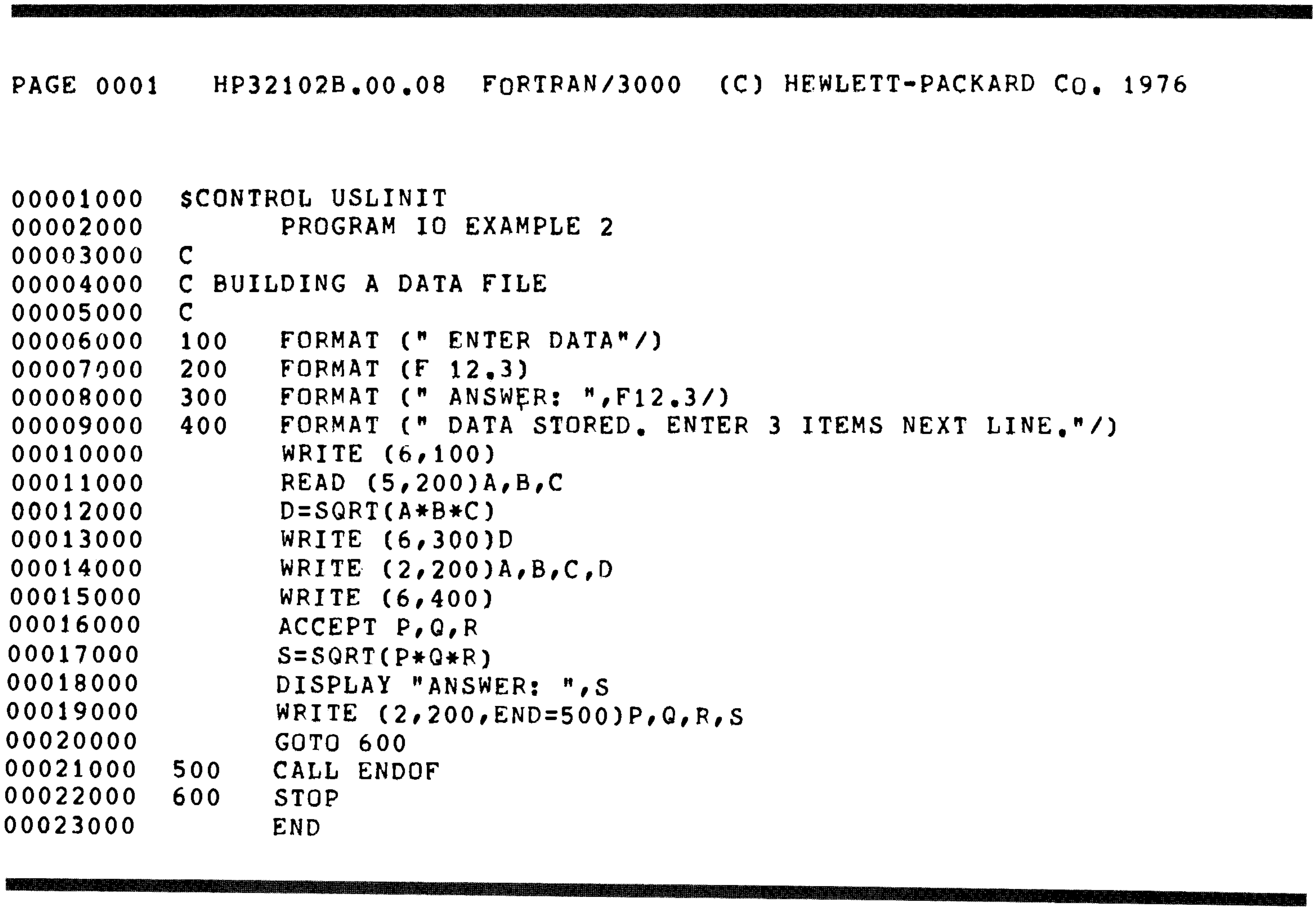
Line 24
calls the subroutine if line 22 encounters an end-of-file.

Lines 27-31
constitute the subroutine, which simply prints out the message quoted in line 28. Since the subroutine is a separate program unit, it will be compiled separately and must have its own END statement. Note that statement labels apply only locally within the subroutinge.

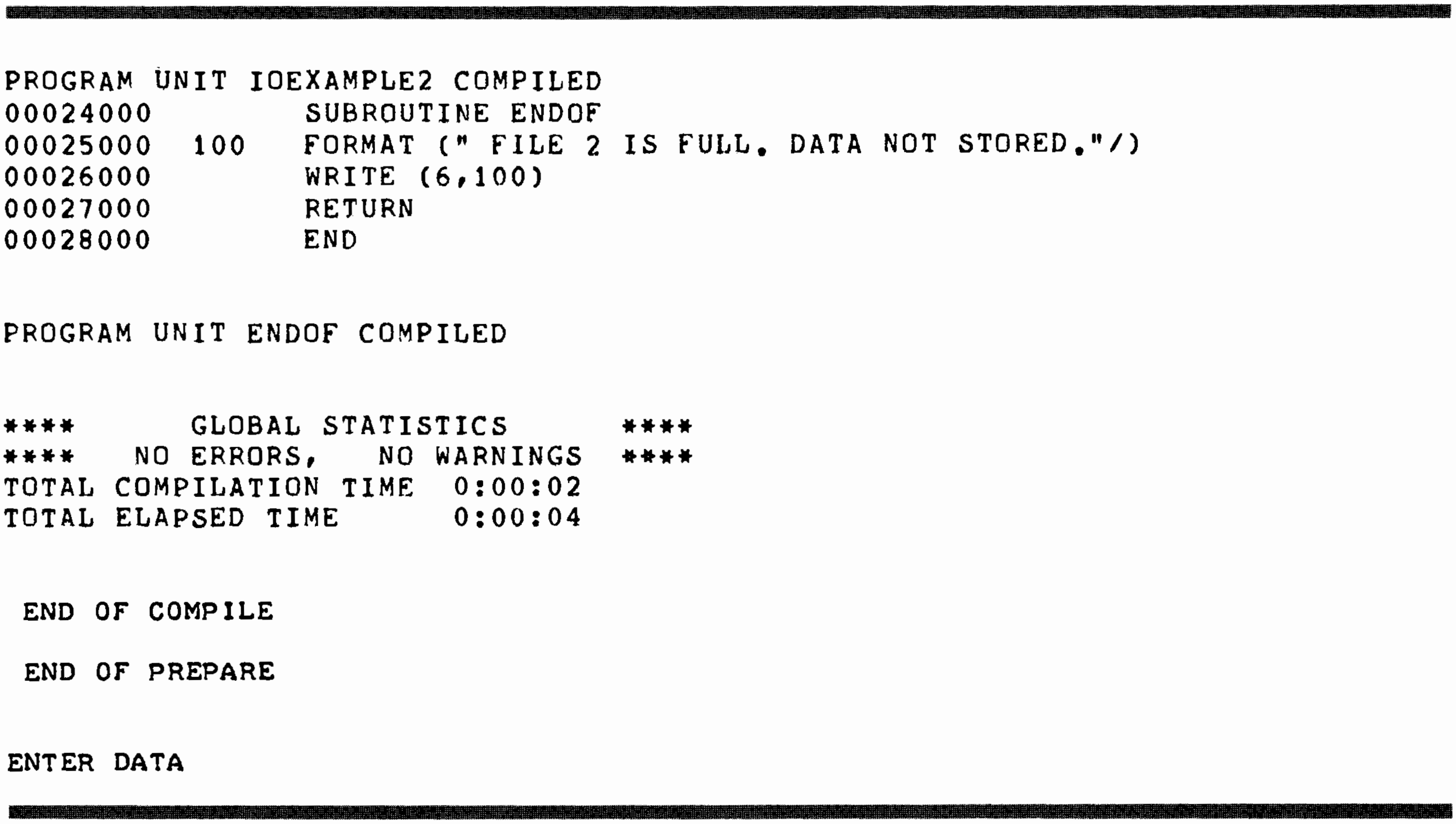
Note the use of GATHERQ. It renumbers the lines.

The new program is now in the source file IODEMO2. Compile, prepare, and execute it with a :FORTGO command. (See printout below.)

Notice that the compilation statistics are printed out for both the main program and the subroutine.



```
PAGE 0001   HP32102B.00.08   FORTRAN/3000   (C) HEWLETT-PACKARD CO. 1976


$CONTROL USLINIT
      PROGRAM IO EXAMPLE 2
C
C BUILDING A DATA FILE
C
100   FORMAT (" ENTER DATA"/)
200   FORMAT (F 12.3)
300   FORMAT (" ANSWER: ",F12.3/)
400   FORMAT (" DATA STORED. ENTER 3 ITEMS NEXT LINE."/)
      WRITE (6,100)
      READ (5,200)A,B,C
      D=SQRT(A*B*C)
      WRITE (6,300)D
      WRITE (2,200)A,B,C,D
      WRITE (6,400)
      ACCEPT P,Q,R
      S=SQRT(P*Q*R)
      DISPLAY "ANSWER: ",S
      WRITE (2,200,END=500)P,Q,R,S
      GOTO 600
500   CALL ENDOF
600   STOP
      END
```



```
PROGRAM UNIT IOEXAMPLE2 COMPILED
00024000        SUBROUTINE ENDOF
00025000  100   FORMAT (" FILE 2 IS FULL. DATA NOT STORED."/)
00026000        WRITE (6,100)
00027000        RETURN
00028000        END


PROGRAM UNIT ENDOF COMPILED


****      GLOBAL STATISTICS      ****
****   NO ERRORS,   NO WARNINGS  ****
TOTAL COMPILATION TIME  0:00:02
TOTAL ELAPSED TIME      0:00:04


 END OF COMPILE

 END OF PREPARE


ENTER DATA
```

When the program stops after printing out ENTER DATA, type in three items of data on separate lines as shown below. Then, after the ? prompt, type three items of data on the same line. The program should then end normally.

```
3456.7
456.78
6.789


ANSWER:       3274.065
DATA STORED. ENTER 3 ITEMS NEXT LINE.

? [illegible]

ANSWER:    3274.06
 END OF PROGRAM
:
```

You now have eight items of data in a 1023-record file. But suppose you need a file that is larger or smaller. In that case you need to build the file yourself, using the :BUILD command, and equate it to the formal file designator (FTN02) with a :FILE command. Do the following.

```
:BUILD DATAFL
:FILE FTN02=DATAFL,OLD
:

:RUN $OLDPASS
```

This builds an empty data file of seven records. This value is chosen here to deliberately invoke an end-of-file on the next run. However, you may set the DISC= parameter to any reasonable value. (The actual maximum is around 300,000 records.)

The :FILE equation links the actual file name DATAFL with the designator FTN02; thus, when you specify "file 2" in your program, the operating system will know which file you really mean. The parameter OLD is required, in this case, so that the operating system will know that the file has already been created (by :BUILD) when your program tries to write to file 2. Thus a new file will not be opened at that time.

Now try another execution of the program. Type RUN $OLDPASS, and enter data as before.

```
ENTER DATA
3456.7
456.78
6.789


ANSWER:       3274.065
DATA STORED. ENTER 3 ITEMS NEXT LINE.

? [illegible]

ANSWER:    3274.06
FILE 2 IS FULL. DATA NOT STORED.

 END OF PROGRAM
:
```

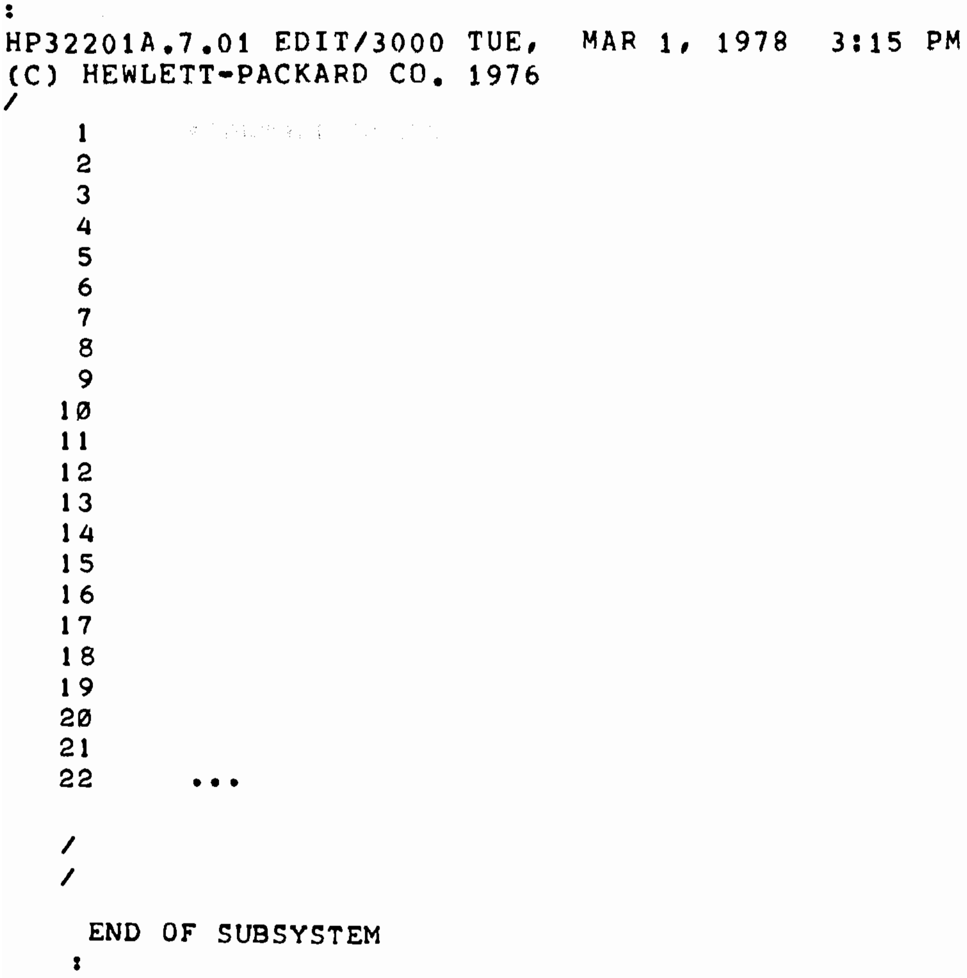
In this run, the WRITE statement in line 19 of the FORTRAN listing was unable to complete its execution since we attempted to store eight records in a seven-record file. This caused a transfer to statement label 500 after the seventh record was stored, resulting in the ENDOF subroutine being called. The subroutine printed out the message FILE 2 IS FULL. DATA NOT STORED.

A point to remember: a :FILE command remains in effect throughout the session. If you want to equate file 2 to some other file, you can enter another :FILE equation; the latest is always the current one in effect. To nullify all :FILE equations for file 2, enter :RESET FTN02.

READING DATA FROM A FILE

This example demonstrates how to read data from a data file. The data file created and filled in the previous exercise is accessed, and an entirely new program is written. Both sequential and direct access are shown.

Call the Editor, then enter the program shown below and store it in a source file called IODEM03.



```
:
HP32201A.7.01 EDIT/3000 TUE,  MAR 1, 1978  3:15 PM
(C) HEWLETT-PACKARD CO. 1976
/
    1
    2
    3
    4
    5
    6
    7
    8
    9
   1Ø
   11
   12
   13
   14
   15
   16
   17
   18
   19
   2Ø
   21
   22      ...

   /
   /

    END OF SUBSYSTEM
   :
```

Some notes on this program:

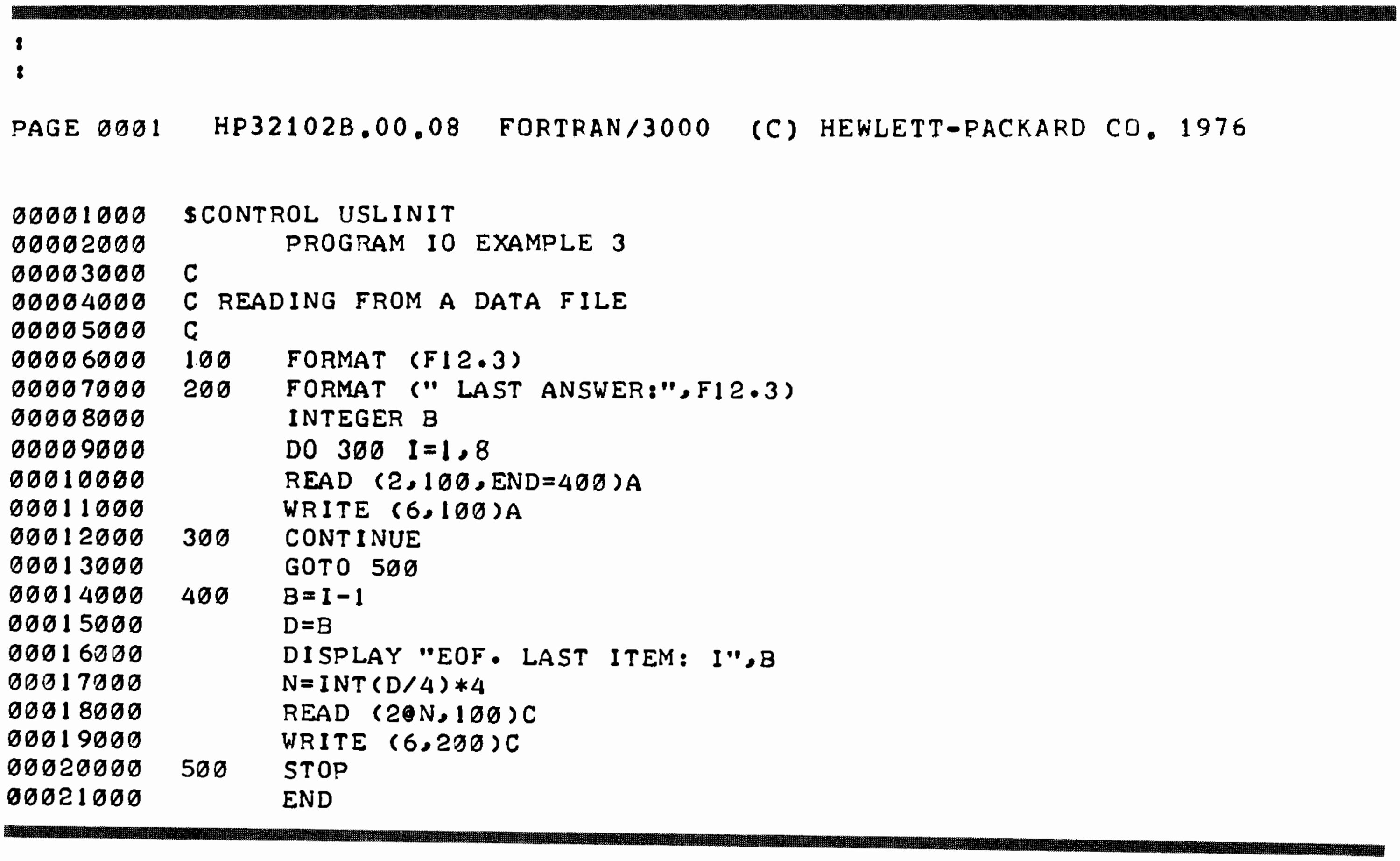
a. Lines 9 through 12 are a do-loop which sequentially reads each record in file 2 and writes each record on the terminal. The READ statement includes a provision to transfer to line 14 if end-of-file is read.

b. Lines 14 and 15 save the item number of the final item read if end-of-file is encountered. Variable B is type integer; D is type real.

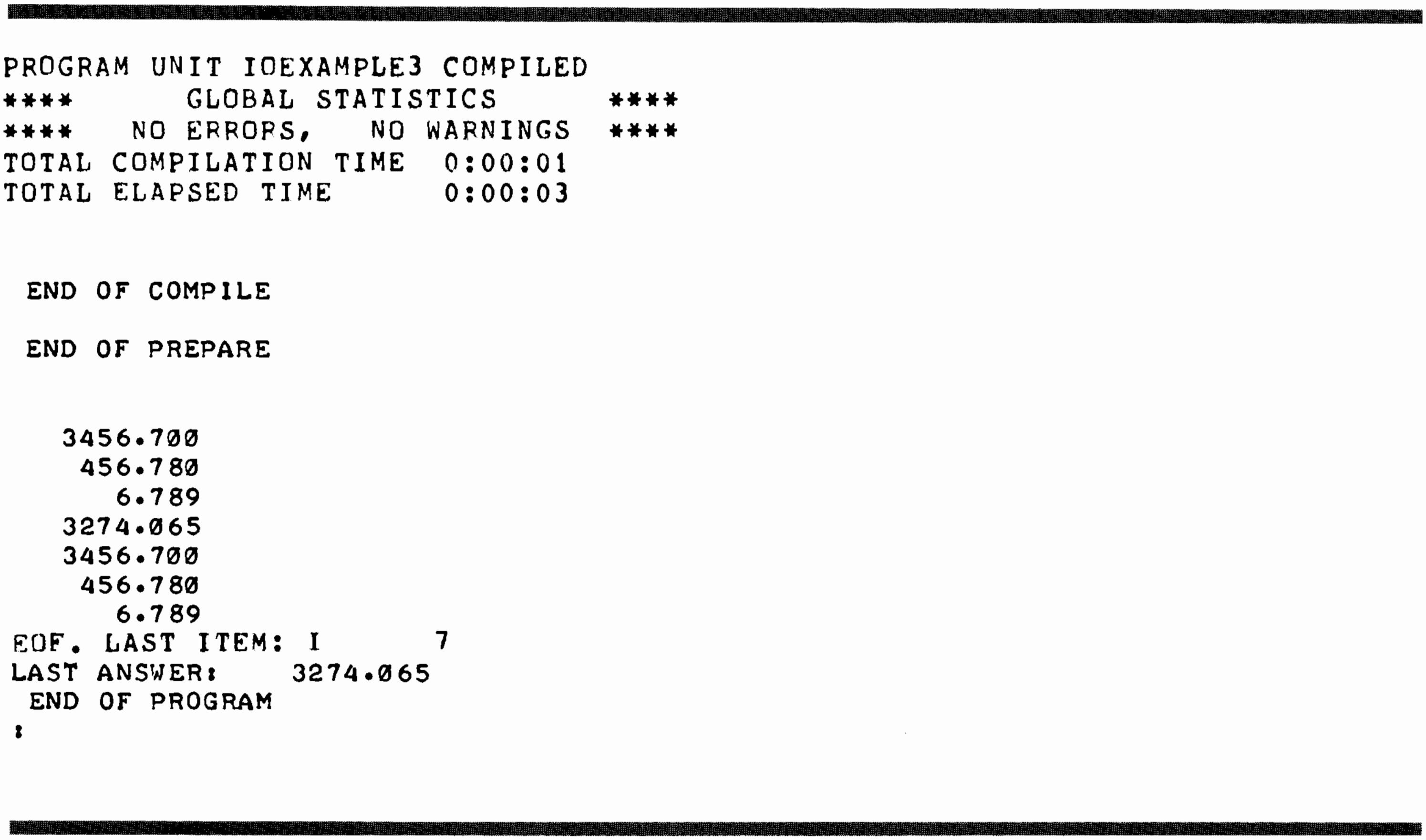
c. Line 17 computes the record number of the last answer value. (In the file DATAFL, every fourth record is an answer value; the preceding three records were entered by you via the terminal.)

d. Line 18 uses direct access to read record "N" in file 2.

Now enter a :FILE equation so the operating system will know that "file 2" is really DATAFL. Then compile, prepare, and execute with a :FORTGO command.



```
:
:

PAGE ØØØ1   HP32102B.00.08   FORTRAN/3000   (C) HEWLETT-PACKARD CO. 1976


ØØØØ1ØØØ  $CONTROL USLINIT
ØØØØ2ØØØ        PROGRAM IO EXAMPLE 3
ØØØØ3ØØØ  C
ØØØØ4ØØØ  C READING FROM A DATA FILE
ØØØØ5ØØØ  C
ØØØØ6ØØØ  1ØØ   FORMAT (F12.3)
ØØØØ7ØØØ  2ØØ   FORMAT (" LAST ANSWER:",F12.3)
ØØØØ8ØØØ        INTEGER B
ØØØØ9ØØØ        DO 3ØØ I=1,8
ØØØ1ØØØØ        READ (2,1ØØ,END=4ØØ)A
ØØØ11ØØØ        WRITE (6,1ØØ)A
ØØØ12ØØØ  3ØØ   CONTINUE
ØØØ13ØØØ        GOTO 5ØØ
ØØØ14ØØØ  4ØØ   B=I-1
ØØØ15ØØØ        D=B
ØØØ16ØØØ        DISPLAY "EOF. LAST ITEM: I",B
ØØØ17ØØØ        N=INT(D/4)*4
ØØØ18ØØØ        READ (2@N,1ØØ)C
ØØØ19ØØØ        WRITE (6,2ØØ)C
ØØØ2ØØØØ  5ØØ   STOP
ØØØ21ØØØ        END
```

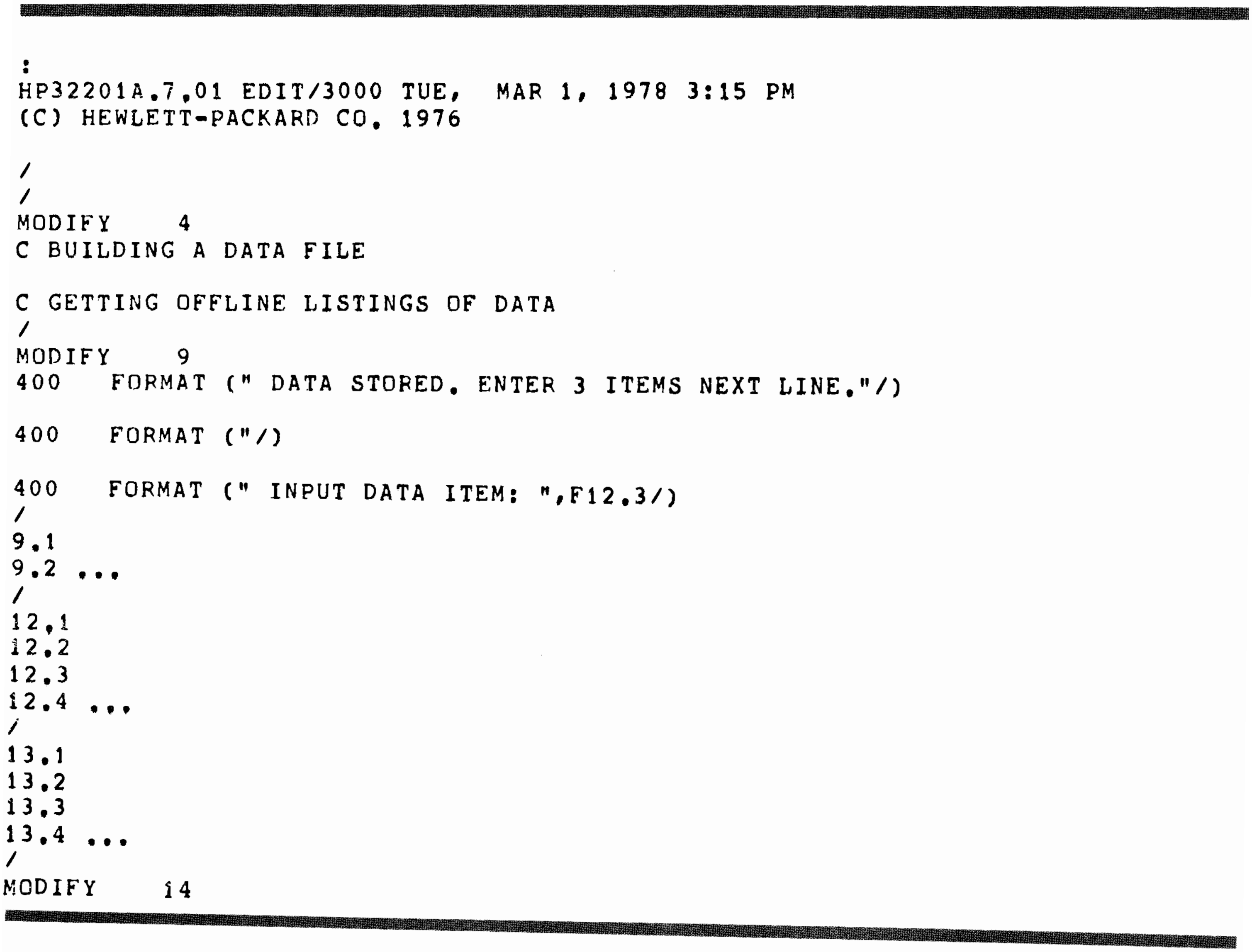


```
PROGRAM UNIT IOEXAMPLE3 COMPILED
****      GLOBAL STATISTICS      ****
****   NO ERRORS,   NO WARNINGS  ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:03


 END OF COMPILE

 END OF PREPARE


   3456.700
    456.780
      6.789
   3274.065
   3456.700
    456.780
      6.789
EOF. LAST ITEM: I       7
LAST ANSWER:     3274.065
 END OF PROGRAM
:
```

As you can see, seven data items are printed out, followed by two lines resulting from an end-of-file branch to statement label 400. These lines state that the last data item read before the end-of-file was item 17, and the last answer value was 3274.065.

offline listing of data

By using a file equation, it is possible to send your results to the line printer. As an example, we use FTN02 to do so. Call the Editor and make the following changes to the IODEM02 file:



```
:
HP32201A.7.01 EDIT/3000 TUE,  MAR 1, 1978 3:15 PM
(C) HEWLETT-PACKARD CO. 1976

/
/
MODIFY    4
C BUILDING A DATA FILE

C GETTING OFFLINE LISTINGS OF DATA
/
MODIFY    9
400   FORMAT (" DATA STORED. ENTER 3 ITEMS NEXT LINE."/)

400   FORMAT ("/)

400   FORMAT (" INPUT DATA ITEM: ",F12.3/)
/
9.1
9.2 ...
/
12.1
12.2
12.3
12.4 ...
/
13.1
13.2
13.3
13.4 ...
/
MODIFY    14
```

```
      WRITE (2,200)A,B,C

      WRITE (2,400)A,B,C
/
14.1
14.2
14.3
14.4 ...
/
NUMBER OF LINES DELETED = 13
/
15
16
17
18
19 ...
/
/
/

END OF SUBSYSTEM
:
```

Note that since we do not need the ACCEPT and DISPLAY statements, and since the ENDOF subroutine is meaningless for a line printer, we delete all entries concerned with them.

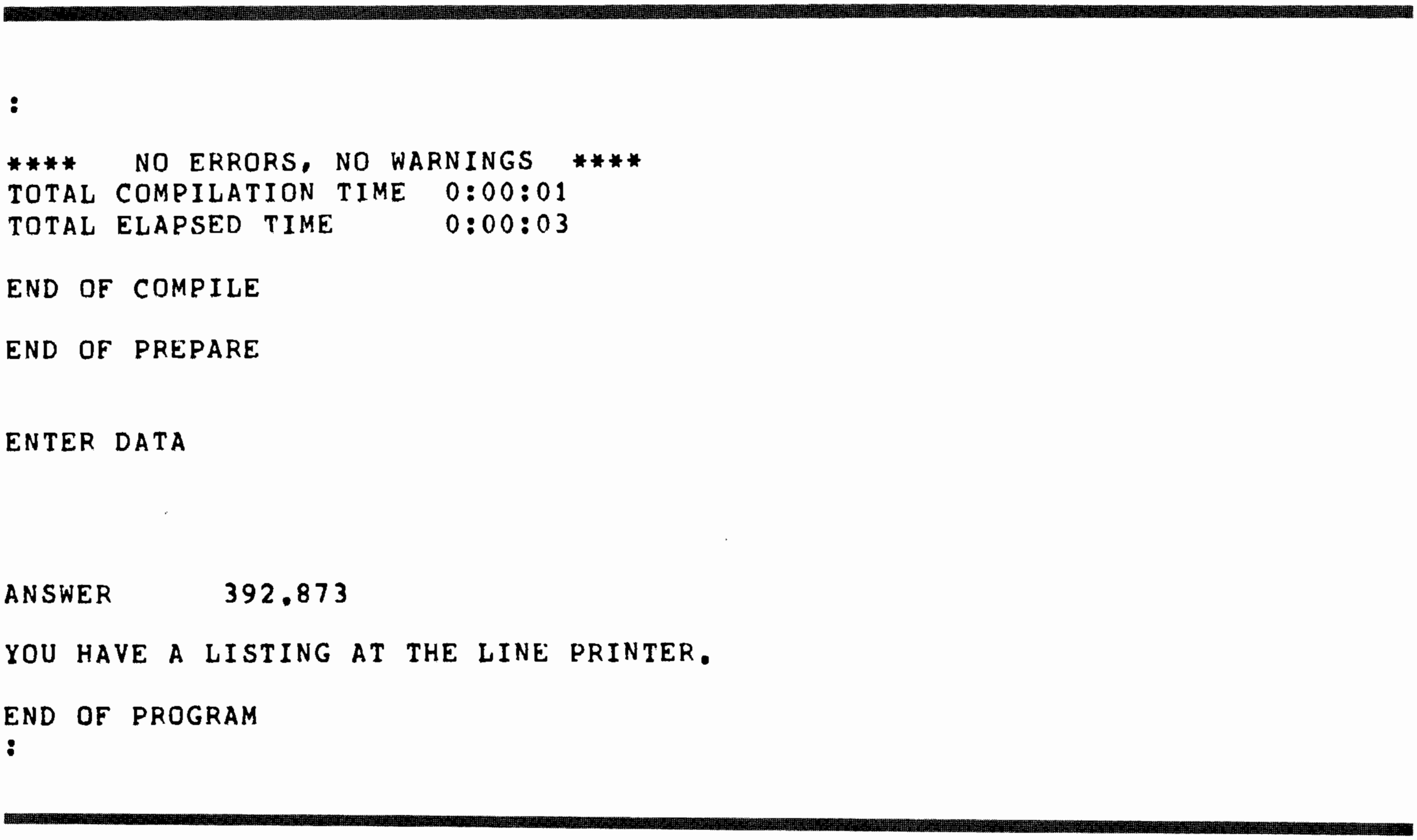
Now, enter the following file equations:

```
:
:
```

The first file equation specifies that FTN02 refers to the line printer. Thus, whenever 2 is specified in a WRITE statement, the data is printed out by the line printer.

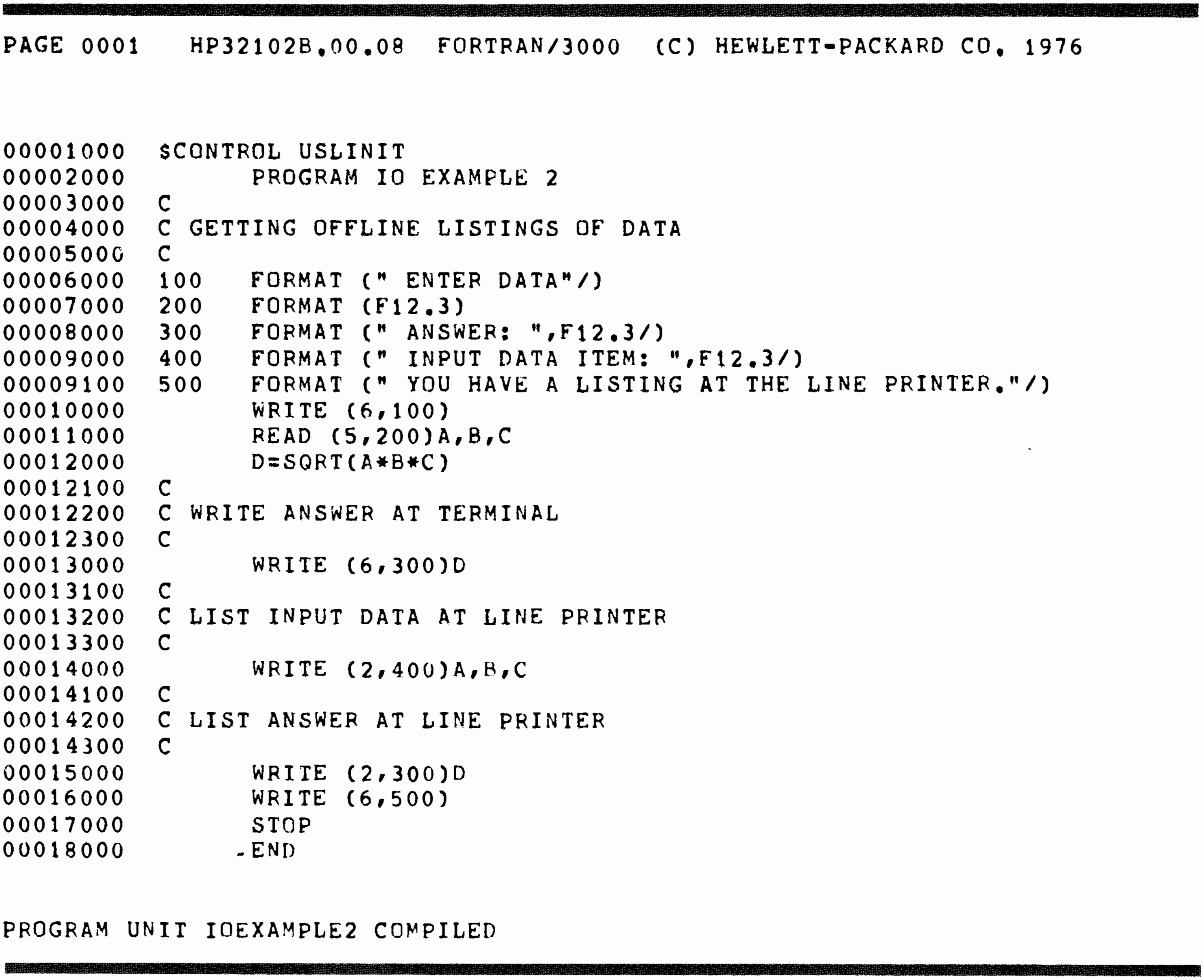
The second file equation should look familiar to you, since we discussed it earlier in this and the Editor section. It is used to list the source file at the line printer. Of course, it is optional; we use it only in order to obtain a listing of the code which generates the output data.

Now, to compile, prepare, and execute the program:



```
:

****   NO ERRORS, NO WARNINGS   ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:03

END OF COMPILE

END OF PREPARE


ENTER DATA



ANSWER      392.873

YOU HAVE A LISTING AT THE LINE PRINTER.

END OF PROGRAM
:
```

The above illustration shows you what is returned to your terminal, and below is the listing which is directed to the line printer.



```
PAGE 0001   HP32102B.00.08  FORTRAN/3000  (C) HEWLETT-PACKARD CO. 1976



$CONTROL USLINIT
      PROGRAM IO EXAMPLE 2
C
C GETTING OFFLINE LISTINGS OF DATA
C
100   FORMAT (" ENTER DATA"/)
200   FORMAT (F12.3)
300   FORMAT (" ANSWER: ",F12.3/)
400   FORMAT (" INPUT DATA ITEM: ",F12.3/)
500   FORMAT (" YOU HAVE A LISTING AT THE LINE PRINTER."/)
      WRITE (6,100)
      READ (5,200)A,B,C
      D=SQRT(A*B*C)
C
C WRITE ANSWER AT TERMINAL
C
      WRITE (6,300)D
C
C LIST INPUT DATA AT LINE PRINTER
C
      WRITE (2,400)A,B,C
C
C LIST ANSWER AT LINE PRINTER
C
      WRITE (2,300)D
      WRITE (6,500)
      STOP
     -END


PROGRAM UNIT IOEXAMPLE2 COMPILED
```

```
PAGE 0002   HEWLETT-PACKARD 32102B.00.08   FORTRAN/3000


****       GLOBAL STATISTICS       ****
****    NO ERRORS,    NO WARNINGS  ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:04
```

```
INPUT DATA ITEM:       34.500

INPUT DATA ITEM:       56.800

INPUT DATA ITEM:      789.098
```

This completes the demonstrations for this section. Purge all unneeded files and, if you're done for this session, log off. (If necessary, use :LISTF to see which files actually exist in your file group.)

```
:
:
:
:
:
:
:
:
:
:
:
```

summary

READ(5,100)A	reads A from terminal keyboard using format statement 100
WRITE (6,200)A	writes A to terminal display unit using format statement 200
ACCEPT P,Q,R	prompts with "?" and accepts three free-field data items from terminal keyboard
DISPLAY "ANDSWER:",S	writes strings and data items to terminal display unit (without format statement)
FORMAT ("ENTER"/)	format statement without data specification requires slash for carriage-return/line-feed; first character in quotes should be blank if not used for carriage control
FORMAT ("ANSWER:",F12.3)	format statement for outputting a string and fixed-point formatted data item
WRITE (8,100)A	writes A to disc file 8; if file does not exist, also creates file of 1023 records
WRITE (8,100,END=500)A	same, but transfers control to statement label 500 if end-of-file is encountered
READ (8, 100,END=400)A	transfers control to statement label 400 if end-of-file encountered on reading file 8
READ (2@5, 100)C	reads record 5 in file 2; record number may be an expression which is truncated to an integer
:BUILD DATAFL;DISC=100	builds a disc file of 100 records called DATAFL (external to FORTRAN)
:FILE FTN02=DATAFL,OLD	equates FORTRAN file 2 to old file (built by :BUILD) called DATAFL
:FILE FTN02;DEV=LP	specifies FORTRAN file 2 be sent to the line printer

extensions from ANSI standard

The following list represents the more significant extensions from the American National Standard Institute's standard for FORTRAN (X3.9 – 1966). A complete list is given in the FORTRAN/3000 reference manual.

Symbolic names consist of as many as 15 characters instead of just 6.

Character-type data can be used in FORTRAN/3000 programs to facilitate string manipulation.

Primaries of different types may be used in creating expressions. In assignment statements, the resulting expression value type is converted to the type of the identifier on the left side of the assignment indicator.

In exponentiation, constructs such as A**B**B are allowed (without the need for parentheses) and can use powers and bases of differing types. No base can be raised to a complex power, however.

Partial-word designators can be used to allow manipulation of the subparts of integer or logical values.

Character-type assignment statements can be used provided the left- and right-hand parts are of type character.

Direct-access files can be referenced in FORTRAN/3000 input/output statements. These statements allow extended format and error recovery capabilities.

Adjustable array declarators can be used for local arrays in subprograms to select different size arrays for each activation of a subprogram, based on passed parameters.

Memory storage space is dependent on the 16-bit word size of the HP 3000. Integer and logical values require one word of computer memory, real values two, double precision four, complex four. Character data uses half-word storage units.

Recursion in function subprogram definition is allowed. The type of actual arguments in a function reference has been expanded. Arguments are all passed by reference rather than value.

section 5
using COBOL/3000

summary of commands introduced in this section

$CONTROL MAP
lists program's symbol names after compilation

$CONTROL NOSOURCE
suppresses source listing during compilation

$CONTROL USLINIT
clears object file before compilation; if used, should be first statement in program

1.4 SELECT COST-FILE ASSIGN TO "COSTFILE".
assigns COBOL file name COST- FILE to system file name COSTFILE

:BUILD COSTFILE; REC=—14,18,F,ASCII
builds an empty system file of 1023 records, 14 characters per record, 18 records per block

:COBOL COBTEST2
compiles source file COBTEST2

:COBOLGO COBTEST1
compiles, prepares, executes source file COBTEST1

:COBOLGO COBTEST3,*LP
compiles, prepares and executes source file COBTEST3; source listing is directed to line printer

:COBOLPREP COBTEST2
compiles and prepares source file COBTEST2; resulting program file is $OLDPASS

:COBOLPREP COBTEST2,P2
compiles and prepares source file COBTEST 2; resulting program file is job-temporary file P2

:FILE COSTFILE;ACC=APPEND
allows adding more data to COSTFILE in next run in this session

:FILE COSTFILE;DEV=LP
equates COSTFILE to line printer

:FILE LP;DEV=LP
used to reference line printer in COBOLGO command

:LISTF COSTFILE,1
displays number of used records, their size and type, and total records allowed for COSTFILE

:PREPRUN $OLDPASS
prepares and executes object file from most recent compilation (where object file was not saved)

:PREPRUN C1
prepares and executes object file C1; assumes $OLDPASS from compilation was saved in C1

summary of commands

:PREP $OLDPASS,P2	prepares object file from most recent compilation into job-temporary file P2
:PREP C2,P3	prepares object file C2 into job temporary file P3
:RESET COSTFILE	nullifies any previous :FILE commands for COSTFILE
:RUN $OLDPASS	execute program file from most recent preparation (where no permanent file was specified)
:RUN P2	executes program file P2
:SAVE $OLDPASS,COBP1	saves most recent compilation or preparation (where no permanent file was specified) in new permanent file COBP1
:SAVE P2	saves file P2 in new permanent file

using COBOL/3000

COBOL/3000 is a compiler language which accepts source statements at the full ANSI COBOL level (with the exception of REPORT WRITER), plus some extensions found only in COBOL/3000. We assume that you are familiar with ANSI COBOL.

The usual sequence of operations is to write a source program using EDIT/3000, compile this program using the COBOL/3000 compiler, and execute it using MPE/3000 Operating System commands.

things to remember

SET FORMAT=COBOL. The Editor includes a special formatting option for COBOL, which you should always use when entering source programs via the Editor. This option, which you obtain by entering the command

```
/SET FORMAT=COBOL
```

will save you the time and trouble of assuring valid sequence numbers and correct column positioning. Specifically, this command establishes the following conditions:

a. The length of the sequence number is six digits instead of eight digits.

b. The sequence numbers are automatically placed in front (columns 1 through 6) instead of at the rear of your source statements. This makes your disc files look like card images.

c. Because the length of the sequence number has been reduced by two, the length of the source line increases from 72 to 74 characters; i.e., the right margin defaults to 74.

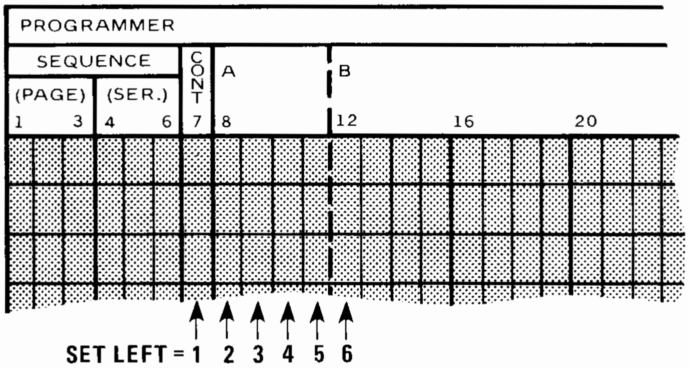
d. The line numbering increment is .1 instead of 1. For example, after a COBOL compile, Editor line numbers 1 and 1.1 will be COBOL sequence number 001000 and 001100.

e. The first column, or left margin, of the Editor line is equivalent to column 7 of a COBOL line. This equivalence (COBOL column number = Editor column number plus 6) holds for any column in an Editor line while /SET FORMAT=COBOL is in effect. Thus, for example, to modify an entry in column 12 of a COBOL line, you specify column 6 of the Editor line.

SET LEFT=. Default condition "e" above may be modified by a command such as

```
SET LEFT=2
```

This moves the left margin from COBOL column 7 to column 8. This means that the position where the carriage or cursor stops is now COBOL column 8. Similarly, a SET LEFT=6 would set the left margin to COBOL column 12, as indicated below.



Always remember where your left margin is set! For example, if you enter LIST ALL and your margin is set to COBOL column 12, only the test in COBOL columns 12 through 74 will be listed. To list COBOL columns 7 through 74, first enter a SET LEFT=1.

Column 7. In COBOL/3000, column 7 is reserved for the continuation character (hyphen), the beginning of a subsystem command (dollar sign), or an indicator that the line is to be treated as a comment (asterisk).

Initialization. The first statement of a COBOL program may be $CONTROL USLINIT. This ensures that the object (USL) file receiving output from the compiler is initialized to an empty condition before compilation starts. (An exception is if you want to compile several program units to the same object file at various times.) The $CONTROL statement starts in column 7.

writing a source program

You write a source program using the Editor the same way as you did in Section 2. That is, you first of all get into the Editor subsystem by typing EDITOR in response to a colon prompt. Then you type ADD and begin typing in your program in response to line number prompts. The only difference is in the special COBOL formatting option (SET commands), described above.

Let's begin. Log on if you have not already, and type the following.

```
:

HP32201A.7.01 EDIT/3000 TUE,  MAR 1, 1978 4:03 PM
(C) HEWLETT-PACKARD CO. 1976
/
/
     1
     1.1
```

The dollar sign will correctly appear in column 7. The next statements, however, have to start in column 8 (Area A of coding form). So here we can use a SET LEFT=2 command. (Alternatively, we could space over one position for each remaining entry.) Terminate the ADD command by typing two slashes, or with a control-Y (Y^c). Then continue as follows.

```
/
/
     1.1
     1.2
     1.3
     1.4
     1.5
     1.6
     1.7
     1.8
     1.9
     2
     2.1
     2.2
     2.3
     2.4
     2.5
```

That completes the first three divisions of the program and the headings for the fourth. The remaining statements are required to begin in COBOL column 12 (Area B of coding form). So we use a SET LEFT=6 command. Terminate the ADD command, enter the SET LEFT command, and enter the rest of the program with a new ADD command.

```
/
/
   2.5
   2.6
   2.7
   2.8
   2.9
   3
   3.1
   3.2
   3.3
   3.4
   3.5
   3.6
   3.7
   3.8
```

Now, to see what the complete program looks like indented, restore the left margin to column 7 (use SET LEFT=1) and type LIST ALL. (Don't forget to terminate ADD.)

```
/
/
   1       $CONTROL USLINIT,SOURCE
   1.1      IDENTIFICATION DIVISION.
   1.2      PROGRAM-ID. COBOL-TEST1.
```

```
 1.3     AUTHOR. YOUR NAME.
 1.4     ENVIRONMENT DIVISION.
 1.5     DATA DIVISION.
 1.6     WORKING-STORAGE SECTION.
 1.7     77  EDIT-FIELD      PIC $Z,ZZ9.99.
 1.8     77  TOTAL-COST      PIC 999V99.
 1.9     77  COST-OF-SALE    PIC 99V99.
 2       77  TAX             PIC 99V99.
 2.1     77  Y-N             PIC X.
 2.2
 2.3     PROCEDURE DIVISION.
 2.4     ENTER-ROUTINE.
 2.5         MOVE ZEROS TO TOTAL-COST.
 2.6         DISPLAY SPACE.
 2.7         DISPLAY "ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT".
 2.8         DISPLAY "(4 DIGITS MAX) INCLUDE LEADING ZEROS!".
 2.9         ACCEPT COST-OF-SALE.
 3           COMPUTE TAX = COST-OF-SALE * .Ø6.
 3.1         ADD COST-OF-SALE, TAX TO TOTAL-COST.
 3.2         MOVE TOTAL-COST TO EDIT-FIELD.
 3.3         DISPLAY "TOTAL COST OF PURCHASE = " EDIT-FIELD.
 3.4         DISPLAY "ARE YOU FINISHED? (Y OR N)".
 3.5         ACCEPT Y-N.
 3.6         IF Y-N = "N" GO TO ENTER-ROUTINE.
 3.7         STOP RUN.
/
```

We expect this program to calculate the tax on the cost of any four-digit amount entered by the terminal operator, and return to the terminal operator the total cost (including tax). A tax rate of 6% is assumed.

The program, now in your work area, may be edited if necessary. When you are satisfied with it, save the source program in a disc file with the KEEP command. Then terminate use of the Editor with an END command.

```
/
/

 END OF SUBSYSTEM
:
```

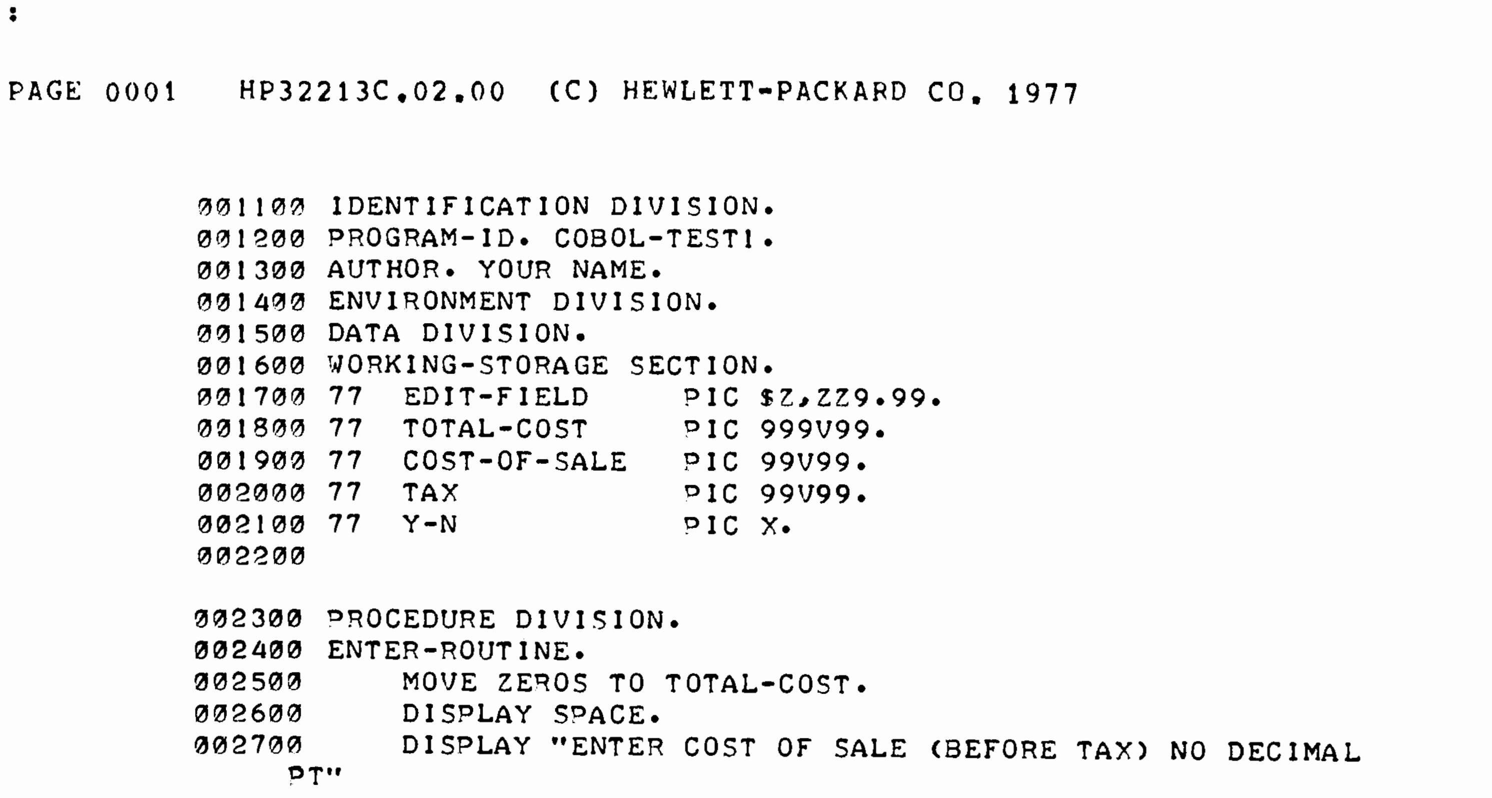
to compile and execute

Just type COBOLGO followed by the file name (COBTEST1) of your source program.

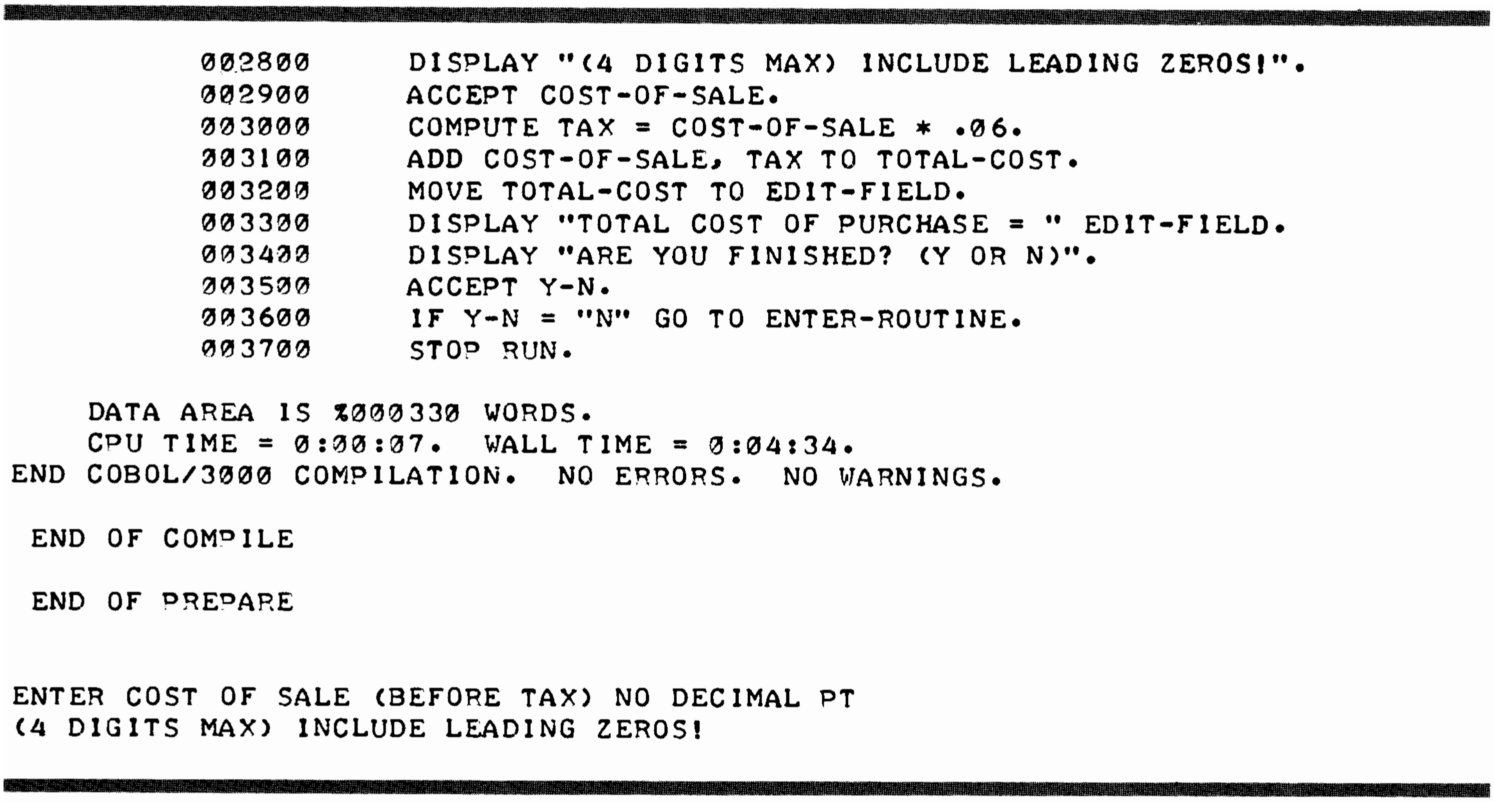
Upon pressing the RETURN key, you will see the following things happen, as illustrated in the sample printout on the next page:

a. A four-digit page number is printed out, followed by the revision identification number of the COBOL/3000 compiler. Page numbering is inserted at intervals appropriate for standard 11-inch page length.

b. Each line of the source program is printed out as it is compiled. This occurs only because we specified a SOURCE parameter in the $CONTROL command; NOSOURCE is the default. Also note that each line is preceded by the line number that was assigned to it by the Editor, except that the number is printed in a special six-digit format. A decimal point is assumed between the third and fourth digits, and leading and trailing zeros are filled in to make six digits.



```
:


PAGE 0001   HP32213C.02.00  (C) HEWLETT-PACKARD CO. 1977


        001100 IDENTIFICATION DIVISION.
        001200 PROGRAM-ID. COBOL-TEST1.
        001300 AUTHOR. YOUR NAME.
        001400 ENVIRONMENT DIVISION.
        001500 DATA DIVISION.
        001600 WORKING-STORAGE SECTION.
        001700 77  EDIT-FIELD     PIC $Z,ZZ9.99.
        001800 77  TOTAL-COST     PIC 999V99.
        001900 77  COST-OF-SALE   PIC 99V99.
        002000 77  TAX            PIC 99V99.
        002100 77  Y-N            PIC X.
        002200

        002300 PROCEDURE DIVISION.
        002400 ENTER-ROUTINE.
        002500     MOVE ZEROS TO TOTAL-COST.
        002600     DISPLAY SPACE.
        002700     DISPLAY "ENTER COST OF SALE (BEFORE TAX) NO DECIMAL
            PT"
```



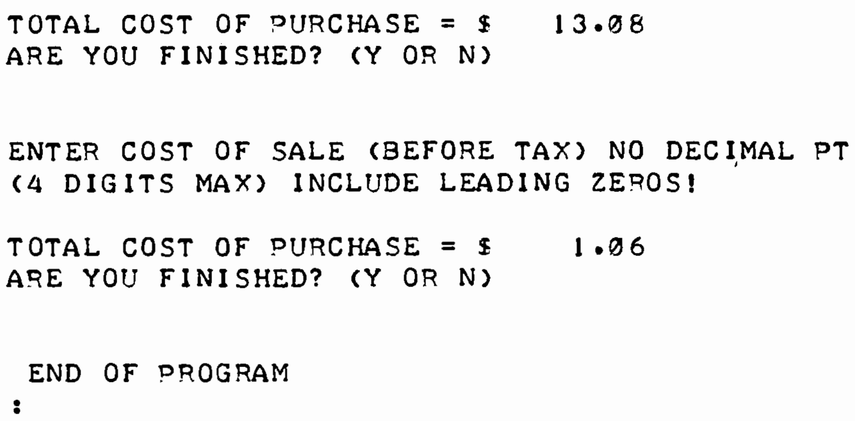
```
          ØØ28ØØ     DISPLAY "(4 DIGITS MAX) INCLUDE LEADING ZEROS!".
          ØØ29ØØ     ACCEPT COST-OF-SALE.
          ØØ3ØØØ     COMPUTE TAX = COST-OF-SALE * .Ø6.
          ØØ31ØØ     ADD COST-OF-SALE, TAX TO TOTAL-COST.
          ØØ32ØØ     MOVE TOTAL-COST TO EDIT-FIELD.
          ØØ33ØØ     DISPLAY "TOTAL COST OF PURCHASE = " EDIT-FIELD.
          ØØ34ØØ     DISPLAY "ARE YOU FINISHED? (Y OR N)".
          ØØ35ØØ     ACCEPT Y-N.
          ØØ36ØØ     IF Y-N = "N" GO TO ENTER-ROUTINE.
          ØØ37ØØ     STOP RUN.

    DATA AREA IS %ØØØ33Ø WORDS.
    CPU TIME = Ø:ØØ:Ø7.  WALL TIME = Ø:Ø4:34.
END COBOL/3ØØØ COMPILATION.  NO ERRORS.  NO WARNINGS.

 END OF COMPILE

 END OF PREPARE


ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT
(4 DIGITS MAX) INCLUDE LEADING ZEROS!
```

c. Next, some statistics about the compilation are printed. The message END COBOL/3000 COMPILATION means that the compilation was successful; if serious errors in your source program prevented successful compilation, you would get the message CHECKED SYNTAX ONLY.

d. The messages END OF COMPILE and END OF PREPARE are next printed. These mean, respectively, that your use of the compiler and the segmenter are successfully terminated. (We will be discussing the role of the segmenter shortly.) If you were unsuccessful, you might get an error message which reads "PROGRAM TERMINATED IN AN ERROR STATE. (CIERR 976)."

e. Finally, you see the first results of executing the sample program. The two lines of a prompt message are printed out, requesting you to enter a four-digit "cost of sale."

To continue with the execution of the program, enter a four-digit value, such as 1234. Respond "N" when asked if you are finished, and enter another value, such as 0100. Then, to terminate the program, respond "Y" the next time you are asked if you are finished.



```
TOTAL COST OF PURCHASE = $     13.Ø8
ARE YOU FINISHED? (Y OR N)


ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT
(4 DIGITS MAX) INCLUDE LEADING ZEROS!

TOTAL COST OF PURCHASE = $      1.Ø6
ARE YOU FINISHED? (Y OR N)


 END OF PROGRAM
:
```

Note that you did not have to press the RETURN key when you entered the data. Whenever the data being entered exactly fills the field defined for it by a PICTURE clause, it is accepted immediately. If you attempt to enter more characters than the field is defined for, ACCEPT takes only enough characters to fill the field.

Also, if the data entered is smaller than the field defined for it in the PICTURE clause, you must tell the program that you are finished entering data. This is done by pressing the RETURN key.

If a field is not completely filled when you enter data, the COBOL compiler does not check the empty digits to see if they contain the proper data. This presents no problem for character data, and is very useful when, for example, you are entering names of people.

However, improper digits in a numeric field cause errors that COBOL/3000 handles by returning an error message, replacing each improper digit with a zero, and continuing the operation. To illustrate, run your program again, and deliberately enter too few digits.

```
:


ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT
(4 DIGITS MAX) INCLUDE LEADING ZEROS!
                                                  (press RETURN key)

*** ERROR 711  ILLEGAL SOURCE DIGIT IN CONVERSION CVAD
*** FIXUP, RESTART ATTEMPTED.
*** ILLEGAL DIGIT REPLACED WITH ZERO.

STATUS REGISTER = %060702    PREGISTER = %000171

SOURCE LENGTH = 4   SOURCE ADDRESS = %000150  DUMP TYPE = ASCII
SOURCE = "230"
TOTAL COST OF PURCHASE= $  24.38
ARE YOU FINISHED? (Y OR N)


END OF PROGRAM
:
```

Since 230 was entered, COBOL/3000 reported that Error 711 had occurred, repaired the error by replacing the improper digit with a zero, showed the original (SOURCE) data, and continued the program, assuming that you meant to enter 2300.

to save the program file

After performing the COBOLGO, the runnable version of the program (or "prepared" program) will be in a system temporary file called $OLDPASS. This can be changed into a permanent file belonging to you by entering:

```
:
:
```

The file name of the permanent file is an arbitrary name, COBP1 in this example. Like all system file names, it must begin with an alphabetic letter, must be no longer than eight characters, and must not already exist.

compilation, preparation, and execution as separate steps

Using :COBOLGO, as you did, simplifies matters for you. All the necessary file manipulations are accomplished automatically with no intervention required from you.

However, there will come times when you may wish to exercise some control over the various phases of program management. For example, :COBOLGO does not allow use of Group or Public library routines, and the compiled object code can neither be accessed nor saved (although the "prepared" program can be). Depending on the type of control you want, you can break down the sequence of program management into any combination of steps that is possible. Six operating system commands give you that control (see Section 3).

The six operating system commands are:

```
:COBOLGO
:COBOLPREP
:COBOL
:PREPRUN
:PREP
:RUN
```

:COBOLGO compiles, prepares, and executes; :COBOLPREP compiles and prepares; :COBOL compiles only; :PREPRUN prepares and executes; and :RUN executes only.

The important characteristics of these commands are listed on this and the next page. We will not, in this manual, demonstrate all of these characteristics; however, it will be useful to be aware of what you can and cannot control. Check the operating system reference manual for the methods of library searches and complete command formats. (And consult your system manager to determine what library routines are available to you.)

But just to get the feel of using these commands, we will now try them all. It's very easy and takes only a few minutes.

First, though, a word about temporary files. You have already used one: $OLDPASS. This is a default file used to receive output code in the compile and prepare steps in the event you do not (or are not permitted to) specify a permanent file. (You will be shown how to specify permanent files in the examples which follow.) If you want to save $OLDPASS in a permanent file, you must do so before another compile or prepare, as this would destroy the existing contents. Job-temporary files, as generated for :COBOLPREP and :PREP, remain in existence for the duration of your session only, but also can be saved if desired (Section 3 discusses how to save such files).

USING :COBOLGO

1. Compiles, prepares, and executes.
2. Object file inaccessible; disappears.
3. Program file may be saved (:SAVE).
4. Programs may *not* reference Group or Public library routines, or Relocatable Library.

USING :COBOLPREP

1. Compiles and prepares.
2. Object file may be saved (add a parameter).
3. Resulting program file is job-temporary.
4. Program may reference library routines available to :RUN command at run time.

USING :COBOL

1. Compiles only.
2. Object file may be saved (:SAVE).
3. Object file must be prepared into program file before execution can occur.

USING :PREPRUN

1. Prepares program file (from object file) and executes same.
2. Program file may be saved (:SAVE).
3. Program may reference routines in Group, Public, or System Library, or Relocatable Library (LIB- or RL=).

USING :PREP

1. Prepares program file (from object file) only.
2. Resulting program file is job-temporary.
3. Program may reference Relocatable Library routines (RL=) plus, at run time, those libraries available to the :RUN command.

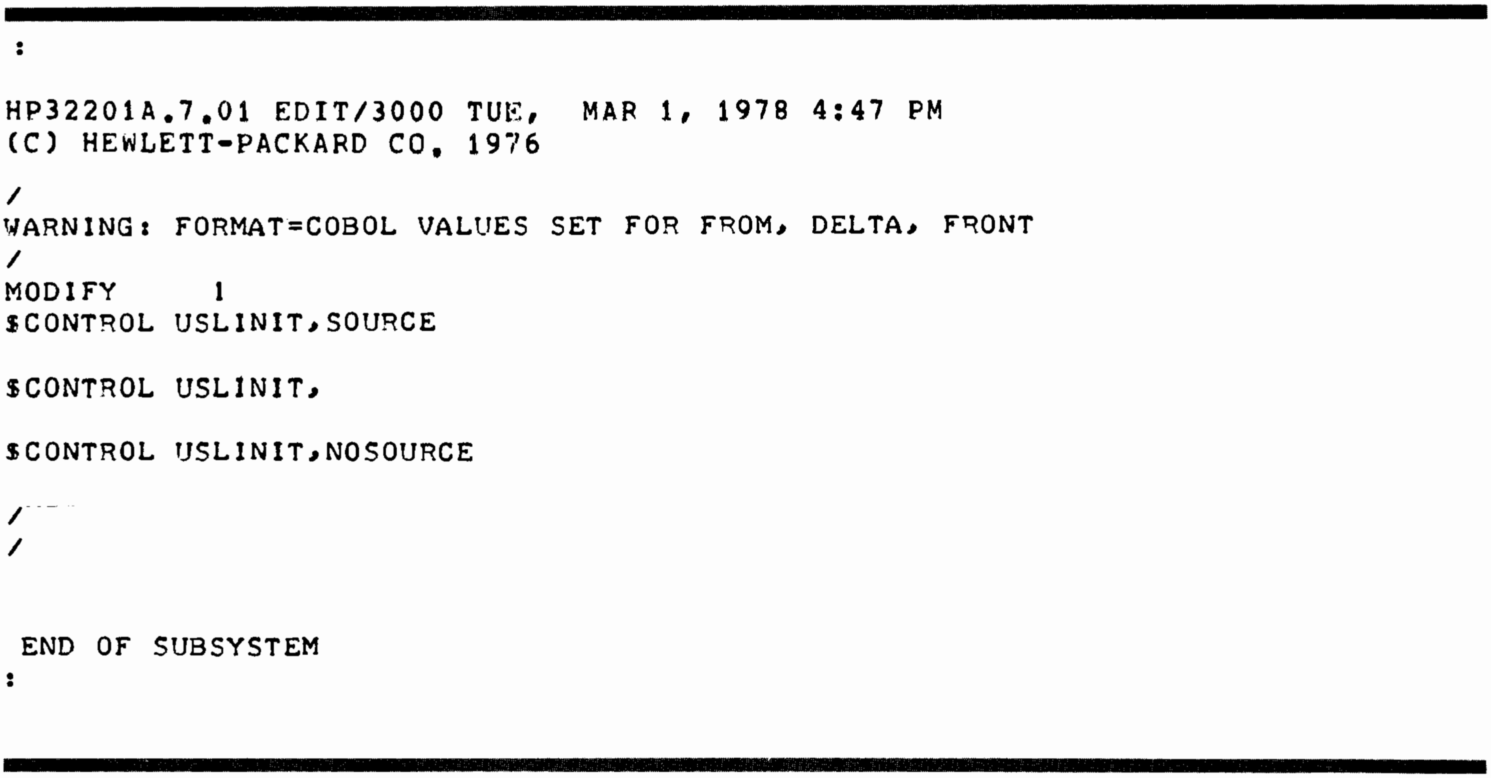
USING :RUN

1. Executes prepared program files only.
2. Program may reference routines in Group, Public, or System Library (LIB=).
3. Program may *not* reference Relocatable Library routines (RL=) unless resolved by :PREP.

COMPILE-PREPARE, THEN EXECUTE

If you did not do the operations on pages 5-7 through 5-9, do so now. You will need the file COBTEST1 for the following operations.

Since the compiler normally prints a source listing on each compilation, we should take advantage of a compiler option that suppresses this listing. (Otherwise you will get very tired of seeing it.) After logging on, respond to the prompts as follows:



```
:

HP32201A.7.01 EDIT/3000 TUE,  MAR 1, 1978 4:47 PM
(C) HEWLETT-PACKARD CO. 1976

/
WARNING: FORMAT=COBOL VALUES SET FOR FROM, DELTA, FRONT
/
MODIFY      1
$CONTROL USLINIT,SOURCE

$CONTROL USLINIT,

$CONTROL USLINIT,NOSOURCE

/
/


 END OF SUBSYSTEM
:
```

The modified source program is now in the disc file COBTEST2. (The original is still in COBTEST1.) Line 1 of the new source file is $CONTROL USLINIT, NOSOURCE. The NOSOURCE parameter states that we do not want a source listing on compilation.

Now proceed. Compile and prepare with a COBOLPREP COBTEST2 command.

```
:


PAGE 0001   HP32213C.02.00  (C) HEWLETT-PACKARD CO. 1977


     DATA AREA IS %ØØØ33Ø WORDS.
     CPU TIME = Ø:ØØ:Ø3.  WALL TIME = Ø:ØØ:37.
END COBOL/3ØØØ COMPILATION.  NO ERRORS.  NO WARNINGS.

 END OF COMPILE

 END OF PREPARE
:
```

Since we didn't specify a permanent program file, the program is now in the temporary file $OLDPASS. Run $OLDPASS as follows:

```
:


ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT
(4 DIGITS MAX) INCLUDE LEADING ZEROS!

TOTAL COST OF PURCHASE = $   45.8Ø
ARE YOU FINISHED? (Y OR N)


 END OF PROGRAM
```

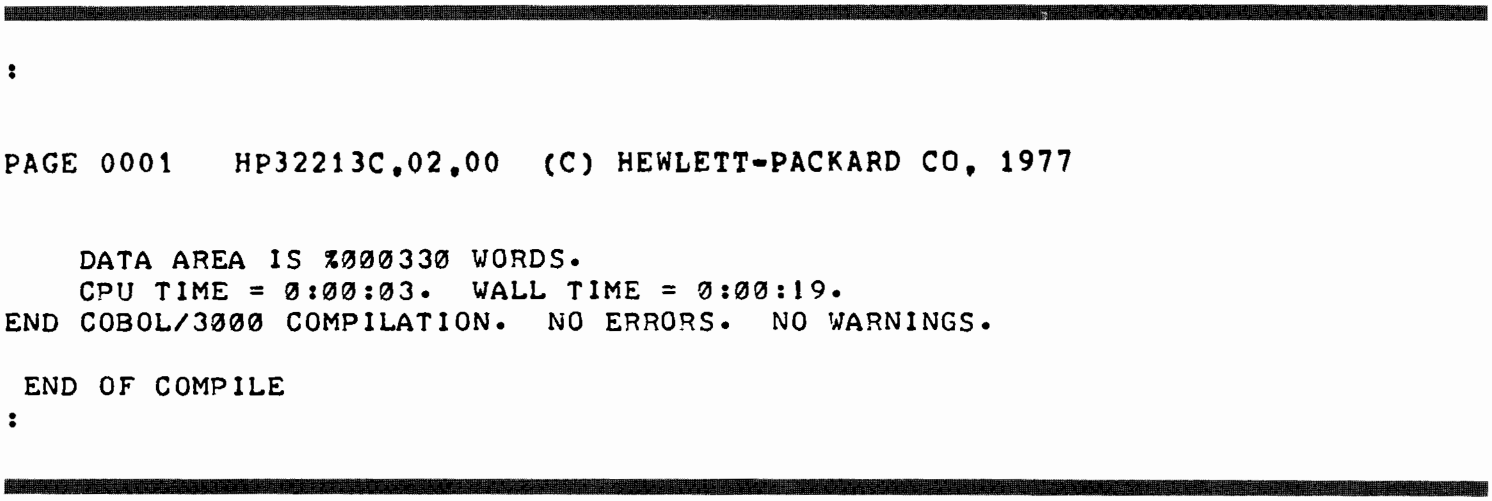
You can save the program file if you like.

```
:
:
```

The above command puts the program in the permanent file P1; you can now RUN P1 at any time.

COMPILE, THEN PREPARE-EXECUTE

Compile by typing COBOL COBTEST2.



```
:


PAGE 0001   HP32213C.02.00  (C) HEWLETT-PACKARD CO. 1977


    DATA AREA IS %000330 WORDS.
    CPU TIME = 0:00:03.  WALL TIME = 0:00:19.
END COBOL/3000 COMPILATION.  NO ERRORS.  NO WARNINGS.

 END OF COMPILE
:
```

The object code is now in $OLDPASS. You could, if you wanted to, save the object code in a permanent file at this time (e.g., :SAVE $OLDPASS,C1). But let's say you do not need a permanent copy of the object code; you can then prepare-execute $OLDPASS directly.

```
:

 END OF PREPARE


ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT
(4 DIGITS MAX) INCLUDE LEADING ZEROS!

TOTAL COST OF PURCHASE = $    1Ø.6Ø
ARE YOU FINISHED? (Y OR N)


 END OF PROGRAM
:
```

(And if you wanted a permanent copy of the program file, you could save $OLDPASS now.)

COMPILE, THEN PREPARE, THEN EXECUTE

Compile by typing COBOL COBTEST2.

```
:


PAGE 0001    HP32213C.02.00   (C) HEWLETT-PACKARD CO. 1977



    DATA AREA IS %ØØØ33Ø WORDS.
    CPU TIME = Ø:ØØ:Ø3.  WALL TIME = Ø:ØØ:19.
END COBOL/3ØØØ COMPILATION.  NO ERRORS.  NO WARNINGS.

 END OF COMPILE
:
```

If you do not need a permanent copy of the object code, you can prepare $OLDPASS directly. (Optionally, you could save the object code now with a :SAVE $OLDPASS,C2.) Use :PREP as follows.

```
:

 END OF PREPARE
:
```

Then, at any time during the session, you can execute the program file P2.

```
:


ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT
(4 DIGITS MAX) INCLUDE LEADING ZEROS!

TOTAL COST OF PURCHASE = $   58.88
ARE YOU FINISHED? (Y OR N)


 END OF PROGRAM
:
```

Note: Unlike :COBOLPREP and :PRERUN, :PREP requires an output file to be specified; there is no default. But remember – the output file so specified is job-temporary. That is, it disappears when you log off. If you want a permanent copy, save the program file (P2) before you log off.

symbol table map

A symbol table map can be obtained at compile time by including a $CONTROL MAP command in the program. This may be combined with other $CONTROL parameters like USLINIT and SOURCE.

A symbol table map is useful for obtaining a complete list of file, data, section, and paragraph names. This map is also usually required when using the MPE/3000 facilities DEBUG and STACKDUMP.

To illustrate, let's get the Editor back and do the following.

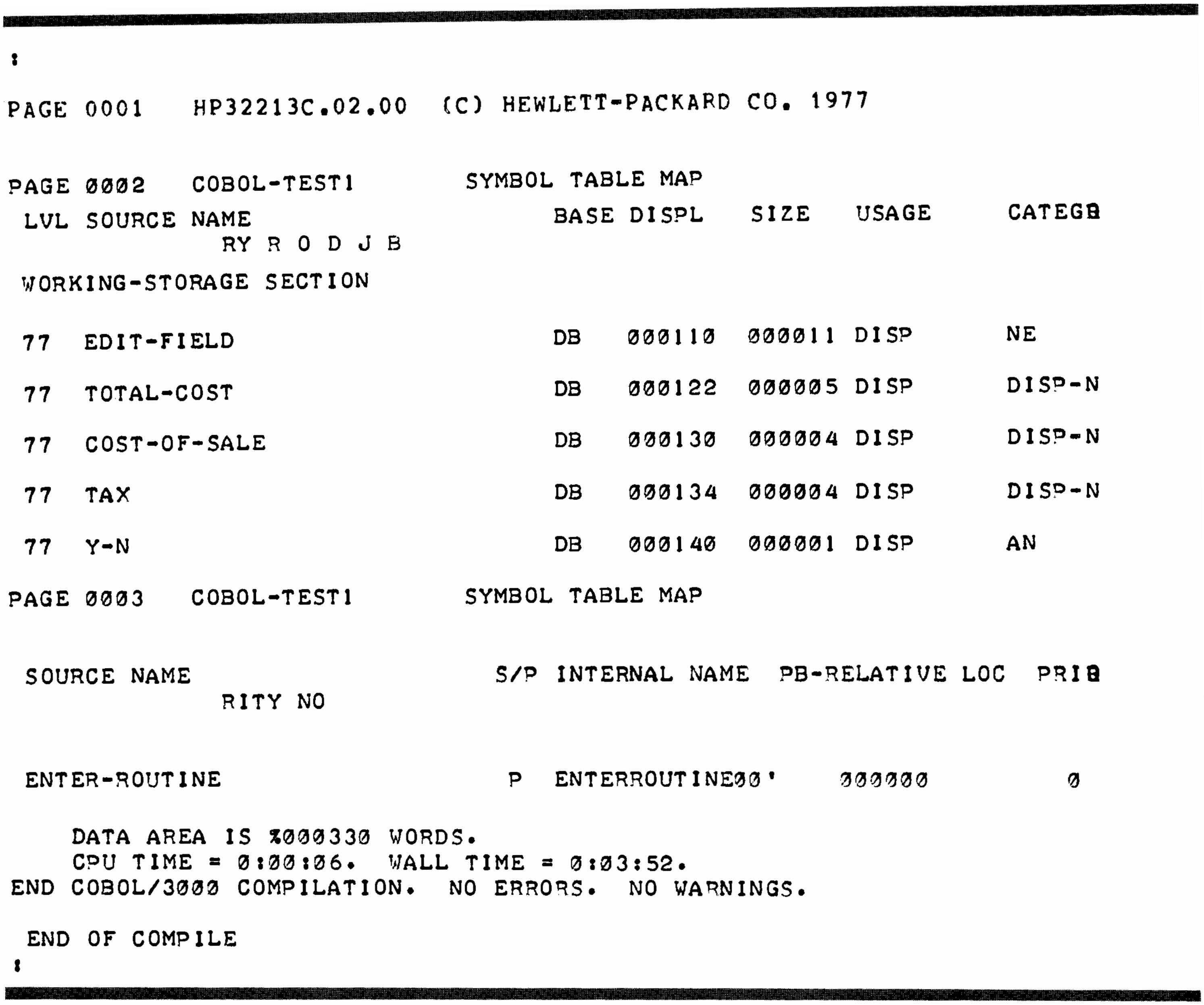
```
:
HP32201A.7.01 EDIT/3000 TUE,  MAR 1, 1978 5:02 PM
(C) HEWLETT-PACKARD CO, 1976

/
WARNING: FORMAT=COBOL VALUES SET FOR FROM, DELTA, FRONT
/
MODIFY      1
$CONTROL USLINIT,NOSOURCE

$CONTROL USLINIT,NOSOURCE,MAP

/
COBTEST2 ALREADY EXISTS - RESPOND YES TO PURGE OLD AND THEN KEEP
PURGE OLD?
/

 END OF SUBSYSTEM
:
```

Now compile with :COBOL COBTEST2 and watch what happens.

Note that the symbol table map has two header lines (SOURCE NAME): one for the file and data names, the other for section and paragraph names.

This particular map shows that we used five data names, which are located in the Working-Storage section. As an example, the first one (EDIT-FIELD) is data-base (DB) relative, with a displacement of 110 octal, a length of 11 characters (octal), a usage of DISPLAY, and a category of Numeric Edited. The next three are all Display Numeric fields, and the last one is an Alpha-Numeric field. Notice there is only one paragraph name listed, ENTER-ROUTINE.

For additional information on the symbol table map and debugging, refer to the COBOL/3000 reference manual and the Using COBOL guide.



```
:

PAGE 0001   HP32213C.02.00  (C) HEWLETT-PACKARD CO. 1977


PAGE Ø002   COBOL-TEST1         SYMBOL TABLE MAP
 LVL SOURCE NAME                      BASE DISPL   SIZE   USAGE     CATEGO
                  RY R O D J B
 WORKING-STORAGE SECTION

 77  EDIT-FIELD                       DB   ØØØ11Ø  ØØØØ11 DISP      NE

 77  TOTAL-COST                       DB   ØØØ122  ØØØØØ5 DISP      DISP-N

 77  COST-OF-SALE                     DB   ØØØ13Ø  ØØØØØ4 DISP      DISP-N

 77  TAX                              DB   ØØØ134  ØØØØØ4 DISP      DISP-N

 77  Y-N                              DB   ØØØ14Ø  ØØØØØ1 DISP      AN
PAGE ØØØ3   COBOL-TEST1         SYMBOL TABLE MAP


 SOURCE NAME                     S/P INTERNAL NAME  PB-RELATIVE LOC  PRIO
                  RITY NO


 ENTER-ROUTINE                    P  ENTERROUTINEØØ'    ØØØØØØ         Ø

    DATA AREA IS %ØØØ33Ø WORDS.
    CPU TIME = Ø:ØØ:Ø6.  WALL TIME = Ø:Ø3:52.
END COBOL/3ØØØ COMPILATION.  NO ERRORS.  NO WARNINGS.

 END OF COMPILE
:
```

summary

$CONTROL USLINIT	clears object file before compilation; should be first statement in program
$CONTROL NOSOURCE	suppresses source listing during compilation
$CONTROL MAP	lists program's symbol names after compilation
:COBOLGO COBTEST1	compiles, prepares, executes source file COBTEST1
:SAVE $OLDPASS,COBP1	saves most recent compilation or preparation (where no permanent file was specified) in new permanent file COBP1
:SAVE P2	saves file P2 in new permanent file
:COBOLPREP COBTEST2	compiles and prepares source file COBTEST2; resulting program file is $OLDPASS
:COBOLPREP COBTEST2,P2	compiles and prepares source file COBTEST2; resulting program file is job-temporary file P2
:RUN $OLDPASS	execute program file from most recent preparation (where no permanent file was specified)
:RUN P2	executes program file P2
:COBOL COBTEST2	compiles source file COBTEST 2
:PREPRUN $OLDPASS	prepares and executes object file from most recent compilation (where object file was not saved)
:PREPRUN C1	prepares and executes object file C1; assumes $OLDPASS from compilation was saved in C1
:PREP $OLDPASS, P2	prepares object file from most recent compilation into job-temporary file P2
:PREP C2,P3	prepares object file C2 into job temporary file P3

using files in COBOL

We have already done some simple terminal input/output. The next examples will demonstrate the basic method of writing data into a file, and reading the data back from the file.

WRITING DATA INTO A FILE

For this example we will modify the previous program. Some of the major additions that have to be made are:

a. A SELECT clause to assign the COBOL file name to the system file name;

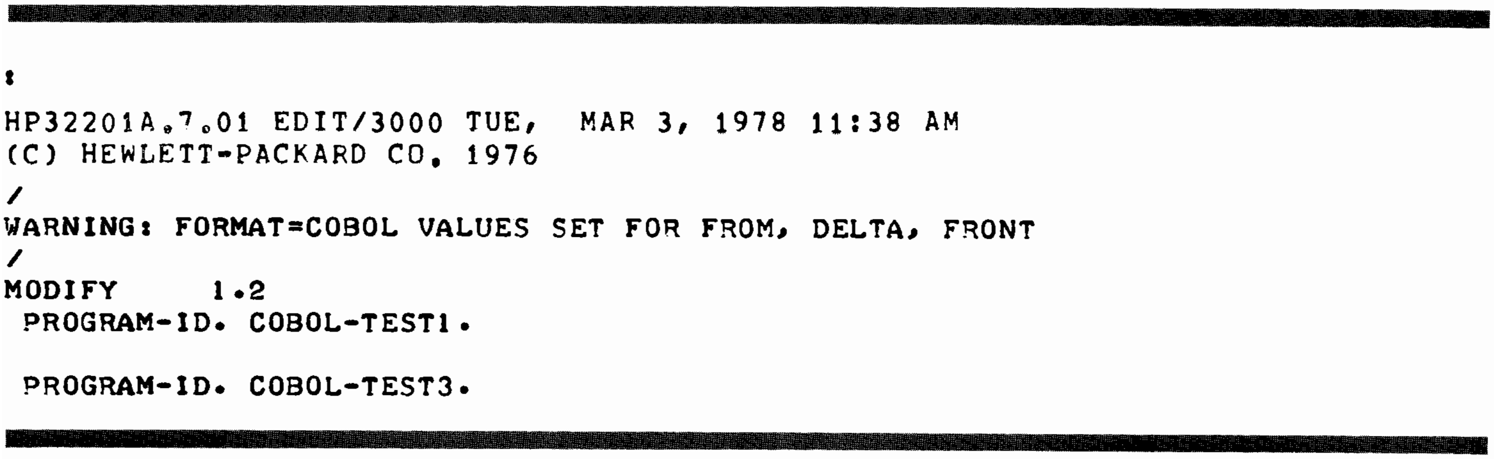
b. a file description (FD) clause to describe the file and the associated record;

c. An OPEN statement for the output file;

d. A WRITE statement to the output file;

e. a CLOSE statement for the output file.

Assuming you still have the source file COBTEST1 from the previous exercise, call the Editor and modify the file as follows. (In each case where three dots are printed, press Y^{c}.)



```
:
HP32201A.7.01 EDIT/3000 TUE,  MAR 3, 1978 11:38 AM
(C) HEWLETT-PACKARD CO. 1976
/
WARNING: FORMAT=COBOL VALUES SET FOR FROM, DELTA, FRONT
/
MODIFY      1.2
 PROGRAM-ID. COBOL-TEST1.

 PROGRAM-ID. COBOL-TEST3.
```

```
/
/
    1.41
    1.42
    1.43
    1.44  ...
/
    1.51
    1.52
    1.53
    1.54
    1.55
    1.56
    1.57  ...
/
    2.31
    2.32
    2.33  ...
/
/
    3.31
    3.32
    3.33
    3.34  ...
/
    3.61
    3.62  ...
/
```

These modifications accomplish the following (see complete listing on next page):

Line 1.2
renames the program to COBOL-TEST3.

Lines 1.41 through 1.43
add the Input-output section.

Lines 1.51 through 1.56
add the File section.

Lines 2.31 through 2.32
add the Open routine.

Line 3.31
moves the current date to the output area DATE-FIELD.

Line 3.32
moves the accumulator TOTAL-COST to the output area T-COST-FIELD.

Line 3.33
writes the cost record to a disc file.

Line 3.61
closes the disc file COST-FILE.

Keep the modified program as COBTEST3 and terminate your use of the Editor.

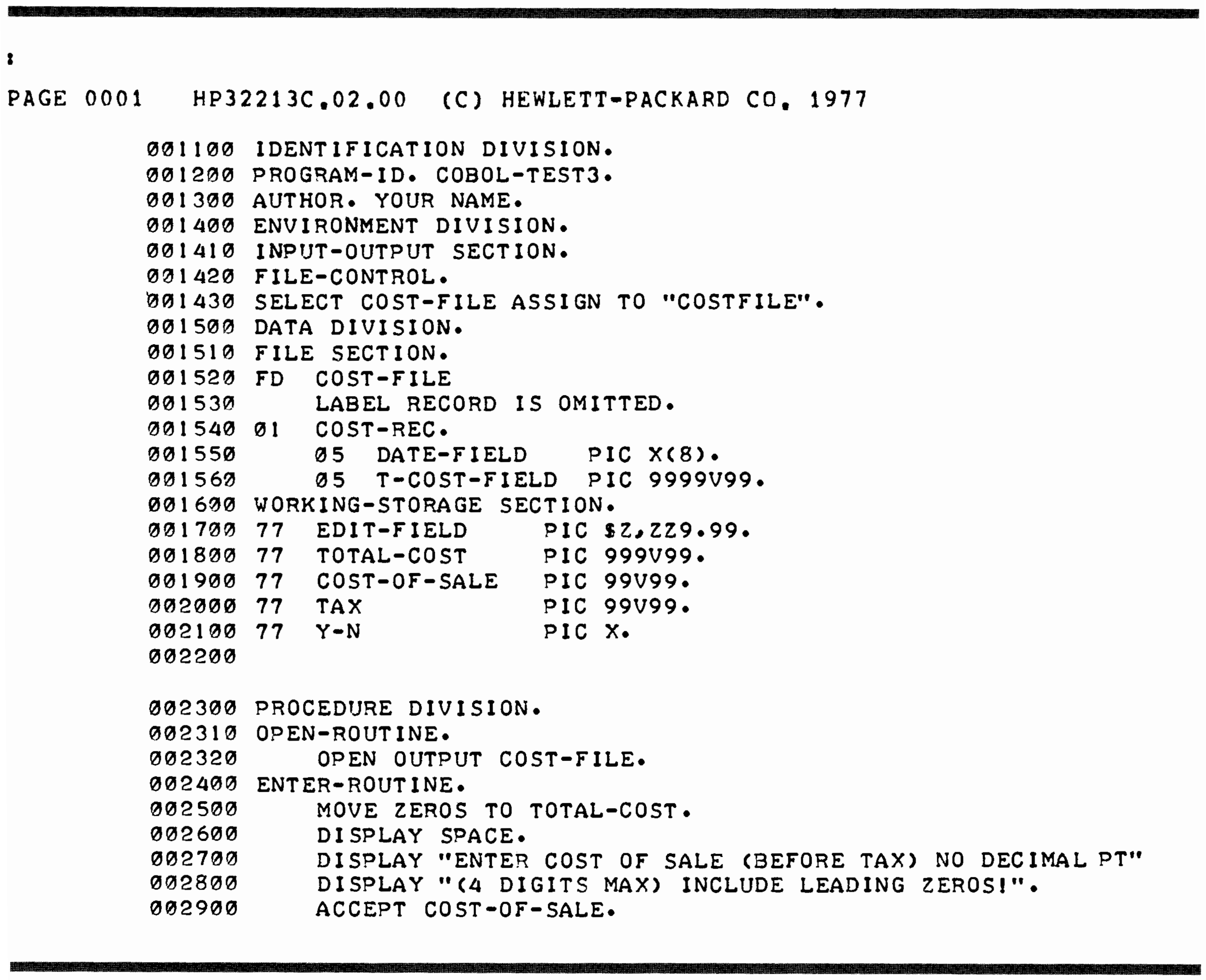
```
/
/

 END OF SUBSYSTEM
:
```

Before you compile and execute this program, you will need to build a file to store the transaction data you will be entering via the terminal.

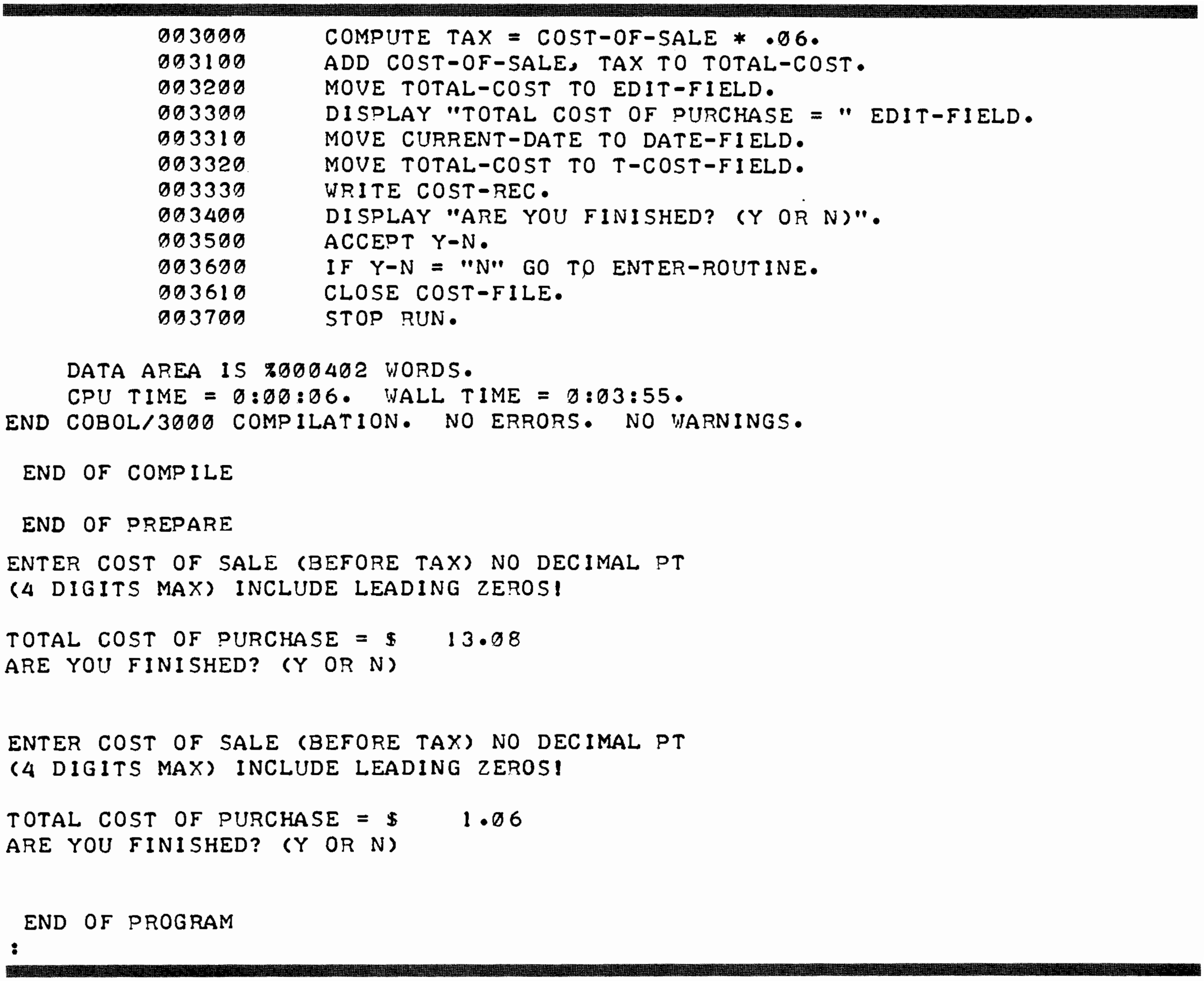
```
:.
:
```

The -14 specifies the record length to be 14 characters; 18 is the blocking factor; F means fixed-length records; and ASCII is the format for storage.

Now compile and run the new program. Enter two values when prompted, replying N or Y after each transaction entered. The total cost for each transaction, including tax, will not only be displayed back to the terminal but will also be written to the file you built, named COSTFILE.



```
:

PAGE 0001   HP32213C.02.00  (C) HEWLETT-PACKARD CO. 1977

        ØØ11ØØ IDENTIFICATION DIVISION.
        ØØ12ØØ PROGRAM-ID. COBOL-TEST3.
        ØØ13ØØ AUTHOR. YOUR NAME.
        ØØ14ØØ ENVIRONMENT DIVISION.
        ØØ141Ø INPUT-OUTPUT SECTION.
        ØØ142Ø FILE-CONTROL.
        ØØ143Ø SELECT COST-FILE ASSIGN TO "COSTFILE".
        ØØ15ØØ DATA DIVISION.
        ØØ151Ø FILE SECTION.
        ØØ152Ø FD  COST-FILE
        ØØ153Ø     LABEL RECORD IS OMITTED.
        ØØ154Ø Ø1  COST-REC.
        ØØ155Ø     Ø5  DATE-FIELD    PIC X(8).
        ØØ156Ø     Ø5  T-COST-FIELD  PIC 9999V99.
        ØØ16ØØ WORKING-STORAGE SECTION.
        ØØ17ØØ 77  EDIT-FIELD     PIC $Z,ZZ9.99.
        ØØ18ØØ 77  TOTAL-COST     PIC 999V99.
        ØØ19ØØ 77  COST-OF-SALE   PIC 99V99.
        ØØ2ØØØ 77  TAX            PIC 99V99.
        ØØ21ØØ 77  Y-N            PIC X.
        ØØ22ØØ

        ØØ23ØØ PROCEDURE DIVISION.
        ØØ231Ø OPEN-ROUTINE.
        ØØ232Ø     OPEN OUTPUT COST-FILE.
        ØØ24ØØ ENTER-ROUTINE.
        ØØ25ØØ     MOVE ZEROS TO TOTAL-COST.
        ØØ26ØØ     DISPLAY SPACE.
        ØØ27ØØ     DISPLAY "ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT"
        ØØ28ØØ     DISPLAY "(4 DIGITS MAX) INCLUDE LEADING ZEROS!".
        ØØ29ØØ     ACCEPT COST-OF-SALE.
```



```
          ØØ3ØØØ      COMPUTE TAX = COST-OF-SALE * .Ø6.
          ØØ31ØØ      ADD COST-OF-SALE, TAX TO TOTAL-COST.
          ØØ32ØØ      MOVE TOTAL-COST TO EDIT-FIELD.
          ØØ33ØØ      DISPLAY "TOTAL COST OF PURCHASE = " EDIT-FIELD.
          ØØ331Ø      MOVE CURRENT-DATE TO DATE-FIELD.
          ØØ332Ø      MOVE TOTAL-COST TO T-COST-FIELD.
          ØØ333Ø      WRITE COST-REC.
          ØØ34ØØ      DISPLAY "ARE YOU FINISHED? (Y OR N)".
          ØØ35ØØ      ACCEPT Y-N.
          ØØ36ØØ      IF Y-N = "N" GO TØ ENTER-ROUTINE.
          ØØ361Ø      CLOSE COST-FILE.
          ØØ37ØØ      STOP RUN.

    DATA AREA IS %ØØØ4Ø2 WORDS.
    CPU TIME = Ø:ØØ:Ø6.  WALL TIME = Ø:Ø3:55.
END COBOL/3ØØØ COMPILATION.  NO ERRORS.  NO WARNINGS.

 END OF COMPILE

 END OF PREPARE
ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT
(4 DIGITS MAX) INCLUDE LEADING ZEROS!

TOTAL COST OF PURCHASE = $   13.Ø8
ARE YOU FINISHED? (Y OR N)


ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT
(4 DIGITS MAX) INCLUDE LEADING ZEROS!

TOTAL COST OF PURCHASE = $    1.Ø6
ARE YOU FINISHED? (Y OR N)


 END OF PROGRAM
:
```

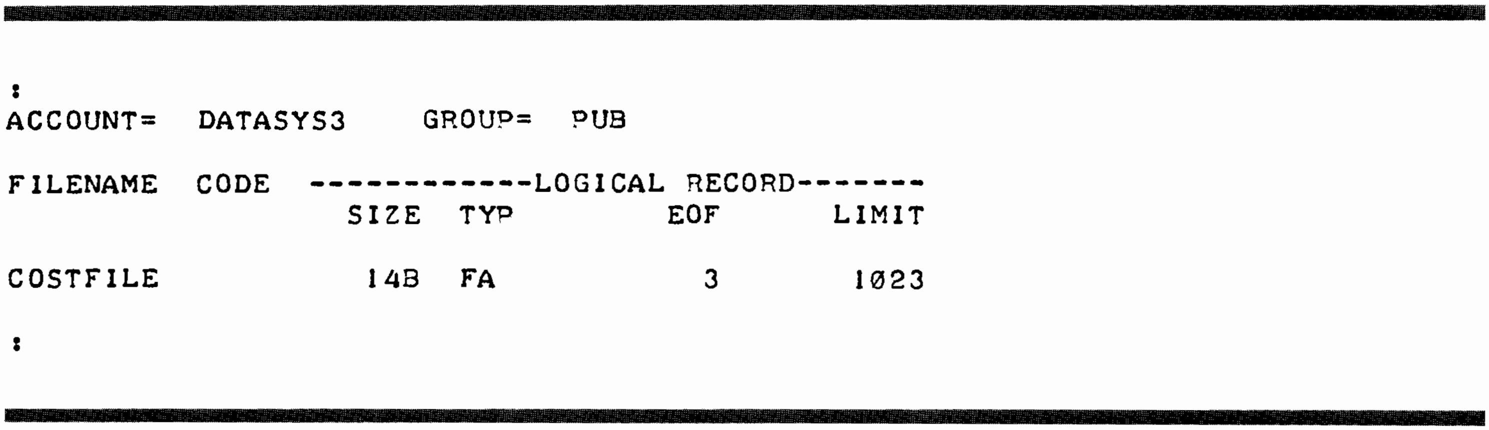
If we wish to append more data to this file in another execution of this program, we must first enter a :FILE command to indicate this.

```
:
:
```

This gives you "append access" to the file. Now try another execution of the program and enter another item of data. This new data will be appended to those items already in the file.

```
:


ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT
(4 DIGITS MAX) INCLUDE LEADING ZEROS!

TOTAL COST OF PURCHASE = $   1Ø.6Ø
ARE YOU FINISHED? (Y OR N)


 END OF PROGRAM
:
```

Now enter LISTF COSTFILE,1 and you will see that the end-of-file (EOF) mark is after record 3. Two records were entered in the first run and one record was entered in the second run.



```
:
ACCOUNT=  DATASYS3    GROUP=  PUB

FILENAME  CODE  ------------LOGICAL RECORD-------
                    SIZE  TYP         EOF      LIMIT

COSTFILE             14B  FA            3       1Ø23

:
```

The specifier ",1" causes all the above information to be printed out. In addition to showing that three records are used, the display also shows that: the record size is 14 bytes; the record type is fixed length (F) ASCII (A); and the maximum number of records allowed is 1023.

offline listing of source and data files

You may wish to obtain a line printer copy of your source file and any data which the program generates. In order to accomplish this, you use two file commands, one for your source file and one for your output data.

When you write to any file, whether it be a disc, tape, cardpunch, or any other possible output device, you must use the SELECT and FD clauses, and the OPEN, WRITE and CLOSE statements. Since COBTEST3 is already set up for an output file, we use it in the following example.

First, issue the FILE commands:

```
:
:
```

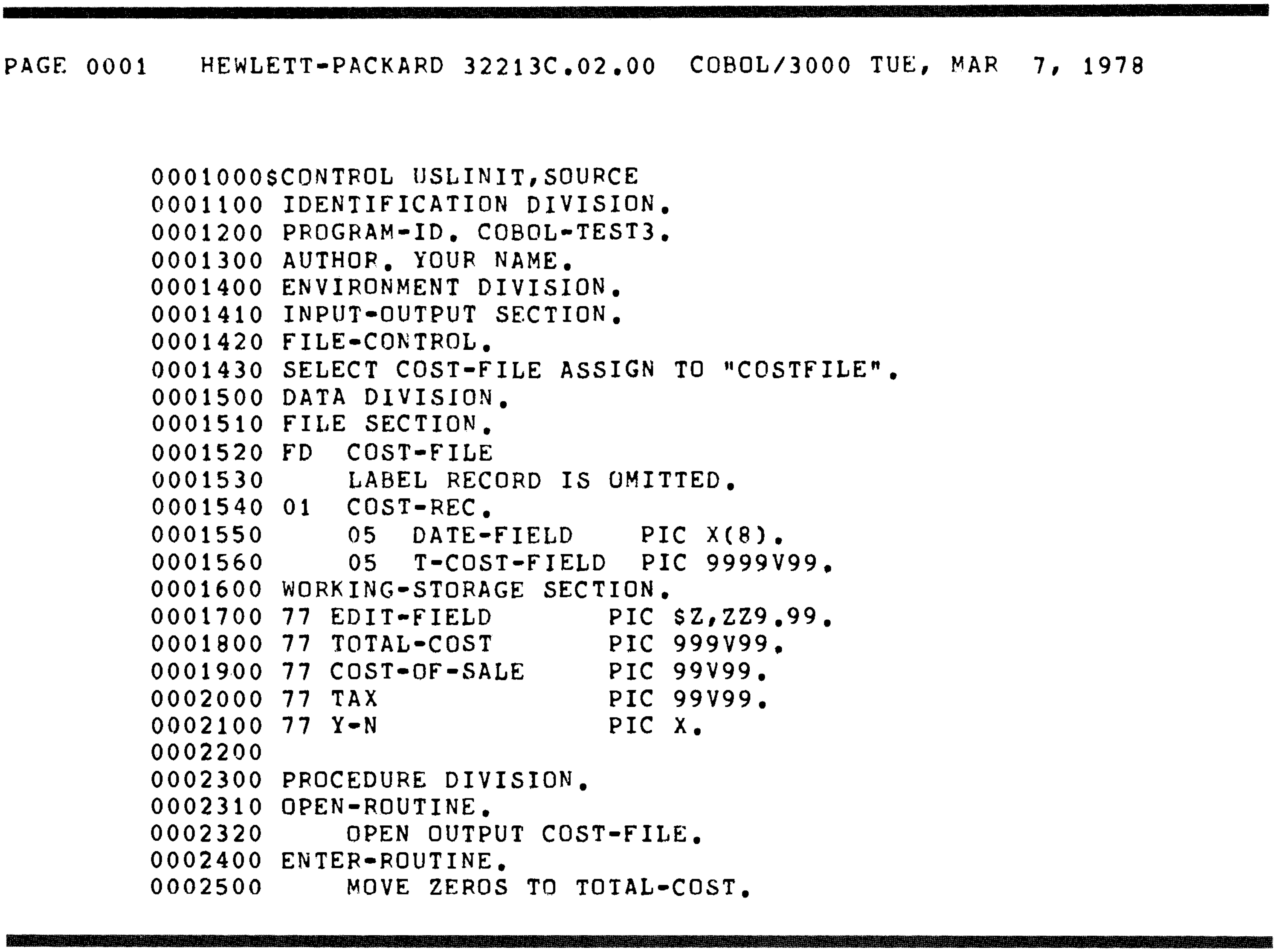
The first equates your data file (in our example, it is COSTFILE) to the line printer; the second is used in the COBOLGO command to tell the operating system that you want your source listing printed by the line printer rather than at your terminal.

Now compile and run COBTEST3, inserting the name LP preceeded by an asterisk, in the COBOLGO command:

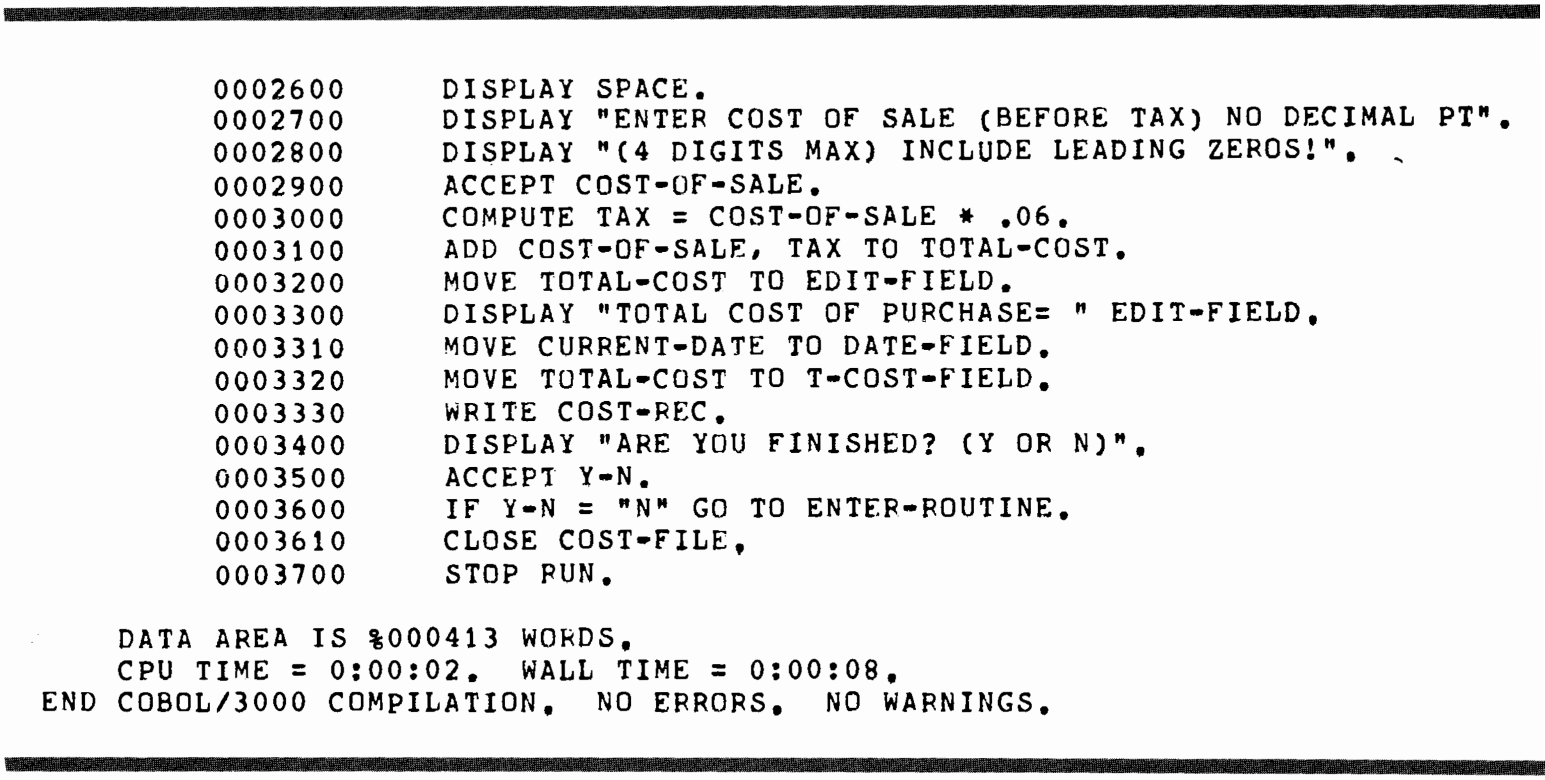
```
:

PAGE 0001   HP32213C.02.00  (C) HEWLETT-PACKARD CO. 1977

    DATA AREA IS %000413 WORDS.
    CPU TIME = 0:00:02.  WALL TIME = 0:00:06.
END COBOL/3000 COMPILATION.  NO ERRORS.  NO WARNINGS.

END OF COMPILE

END OF PREPARE
```

```
ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT
(4 DIGITS MAX) INCLUDE LEADING ZEROS!

TOTAL COST OF PURCHASE = $   11.77
ARE YOU FINISHED? (Y OR N)

ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT
(4 DIGITS MAX) INCLUDE LEADING ZEROS!

TOTAL COST OF PURCHASE = $   23.55
ARE YOU FINISHED? (Y OR N)

ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT
(4 DIGITS MAX) INCLUDE LEADING ZEROS!

TOTAL COST OF PURCHASE = $   35.32
ARE YOU FINISHED? (Y OR N)

ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT
(4 DIGITS MAX) INCLUDE LEADING ZEROS!

TOTAL COST OF PURCHASE = $   47.10
ARE YOU FINISHED? (Y OR N)

ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT
(4 DIGITS MAX) INCLUDE LEADING ZEROS!

TOTAL COST OF PURCHASE = $   58.88
ARE YOU FINISHED? (Y OR N)


 END OF PROGRAM
:
```

The above example shows you what is returned to your terminal, including your input; below is the listing which is directed to the line printer.



```
PAGE 0001    HEWLETT-PACKARD 32213C.02.00  COBOL/3000 TUE, MAR  7, 1978


        0001000$CONTROL USLINIT,SOURCE
        0001100 IDENTIFICATION DIVISION.
        0001200 PROGRAM-ID. COBOL-TEST3.
        0001300 AUTHOR. YOUR NAME.
        0001400 ENVIRONMENT DIVISION.
        0001410 INPUT-OUTPUT SECTION.
        0001420 FILE-CONTROL.
        0001430 SELECT COST-FILE ASSIGN TO "COSTFILE".
        0001500 DATA DIVISION.
        0001510 FILE SECTION.
        0001520 FD  COST-FILE
        0001530     LABEL RECORD IS OMITTED.
        0001540 01  COST-REC.
        0001550     05  DATE-FIELD    PIC X(8).
        0001560     05  T-COST-FIELD  PIC 9999V99.
        0001600 WORKING-STORAGE SECTION.
        0001700 77 EDIT-FIELD        PIC $Z,ZZ9.99.
        0001800 77 TOTAL-COST        PIC 999V99.
        0001900 77 COST-OF-SALE      PIC 99V99.
        0002000 77 TAX               PIC 99V99.
        0002100 77 Y-N               PIC X.
        0002200
        0002300 PROCEDURE DIVISION.
        0002310 OPEN-ROUTINE.
        0002320     OPEN OUTPUT COST-FILE.
        0002400 ENTER-ROUTINE.
        0002500     MOVE ZEROS TO TOTAL-COST.
```



```
        0002600       DISPLAY SPACE.
        0002700       DISPLAY "ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT".
        0002800       DISPLAY "(4 DIGITS MAX) INCLUDE LEADING ZEROS!".
        0002900       ACCEPT COST-OF-SALE.
        0003000       COMPUTE TAX = COST-OF-SALE * .06.
        0003100       ADD COST-OF-SALE, TAX TO TOTAL-COST.
        0003200       MOVE TOTAL-COST TO EDIT-FIELD.
        0003300       DISPLAY "TOTAL COST OF PURCHASE= " EDIT-FIELD.
        0003310       MOVE CURRENT-DATE TO DATE-FIELD.
        0003320       MOVE TOTAL-COST TO T-COST-FIELD.
        0003330       WRITE COST-REC.
        0003400       DISPLAY "ARE YOU FINISHED? (Y OR N)".
        0003500       ACCEPT Y-N.
        0003600       IF Y-N = "N" GO TO ENTER-ROUTINE.
        0003610       CLOSE COST-FILE.
        0003700       STOP RUN.

    DATA AREA IS %000413 WORDS.
    CPU TIME = 0:00:02.  WALL TIME = 0:00:08.
END COBOL/3000 COMPILATION.  NO ERRORS.  NO WARNINGS.
```

A second listing is generated for your output data. Below is the result of the listing, without the header information that normally appears at the top of the page.

```
03/06/78001177
03/06/78002355
03/06/78003532
03/06/78004710
03/06/78005888
```

READING DATA FROM A FILE

This example demonstrates how to read data from the previously created data file, COSTFILE. The previous COBOL source file COBTEST3 must first be modified to allow it to read data from the file and to display the data. In addition, the program will display the grand total after printing out the separate transactions.

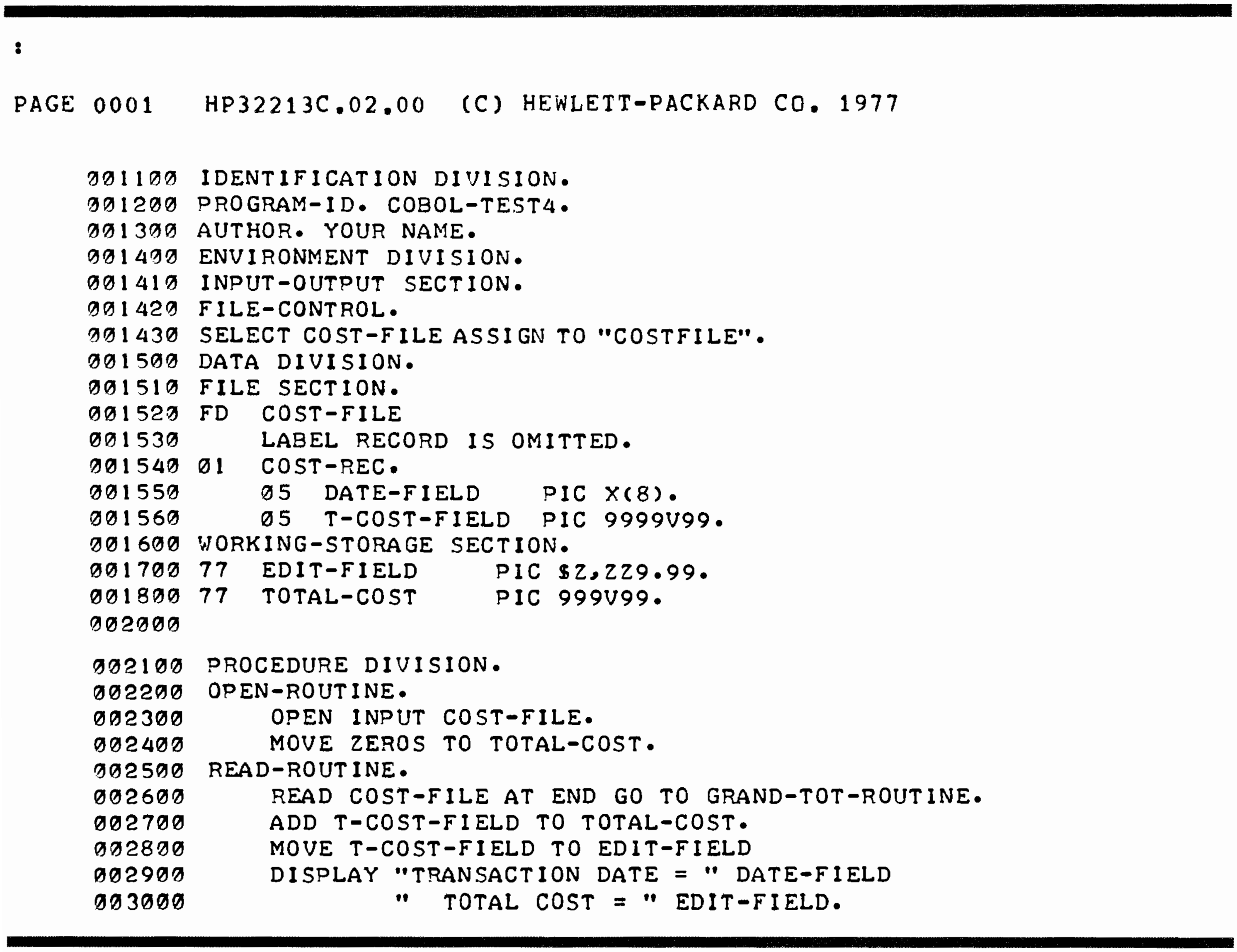
Call the Editor, then modify the program as shown below. We begin by deleting three data names, which are unnecessary in this example, and the entire ENTER-ROUTINE, which is the remainder of the old program. Then add a complete new procedure division. Keep the new source program as COBTEST4.

```
:

HP32201A.7.01 EDIT/3000 TUE,  MAR 3, 1978 12:13 PM
(C) HEWLETT-PACKARD CO. 1976
/
WARNING: FORMAT=COBOL VALUES SET FOR FROM, DELTA, FRONT
/
NUMBER OF LINES DELETED = 25
/
/
MODIFY      1.2
PROGRAM-ID. COBOL-TEST3.

PROGRAM-ID. COBOL-TEST4.

/
    2
    2.1
    2.2
    2.3
    2.4
    2.5
    2.6    ...
/
/
    2.6
    2.7
    2.8
    2.9
    3
    3.1
    3.2    ...
/
/
    3.2
    3.3    ...
/
/
    3.3
    3.4
    3.5                 ◄———— 20 Spaces ————►
    3.6
    3.7
    3.8
    3.9    ...
/
/


 END OF SUBSYSTEM
:
```

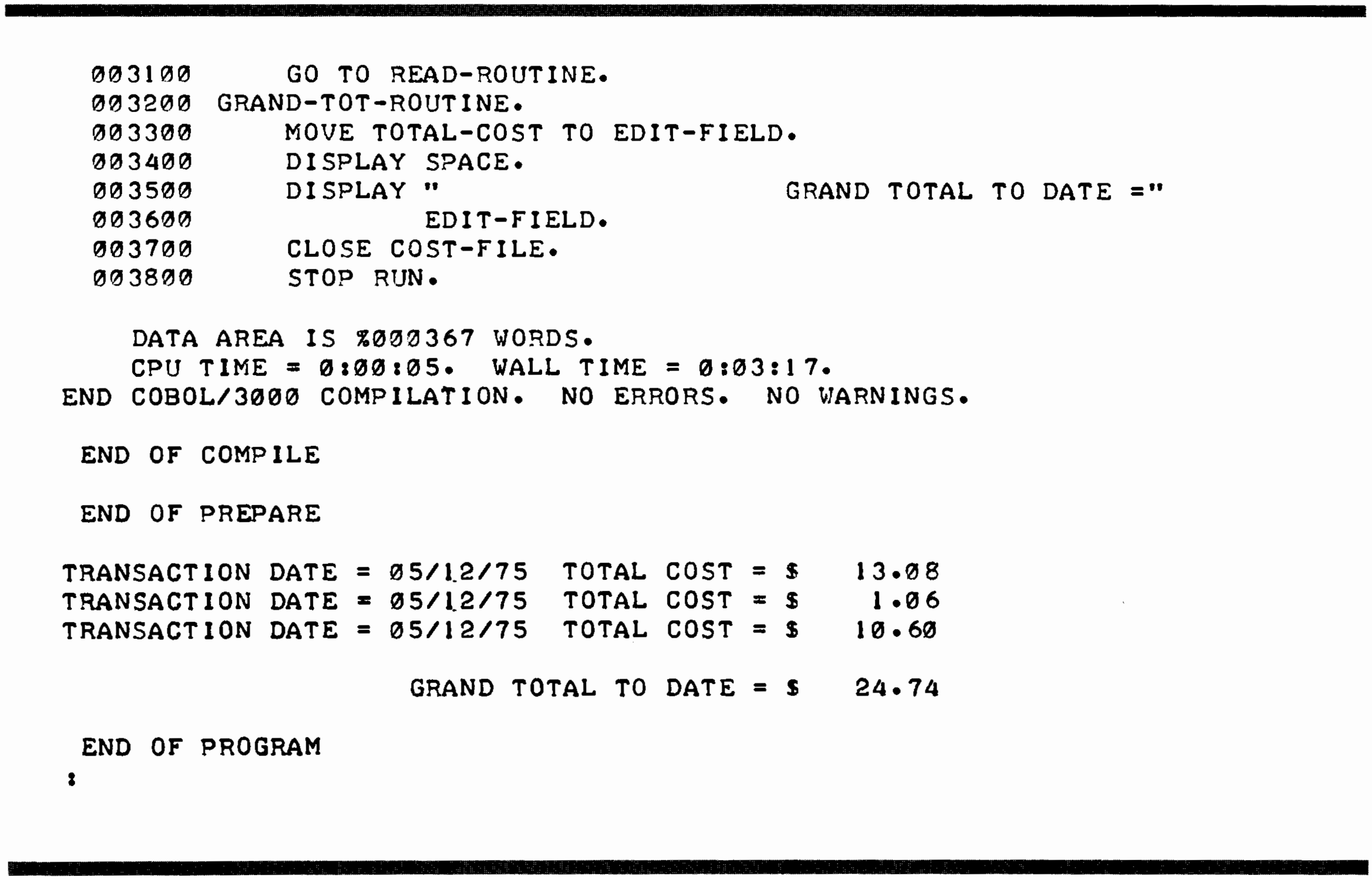
If you haven't logged off since the previous exercise, you will have to enter a RESET command to nullify the :FILE command which specified COSTFILE as a line printer file.

```
:
:
```

The new program is now in the source file COBTEST4. Compile, prepare, and execute it with a COBOLGO command.



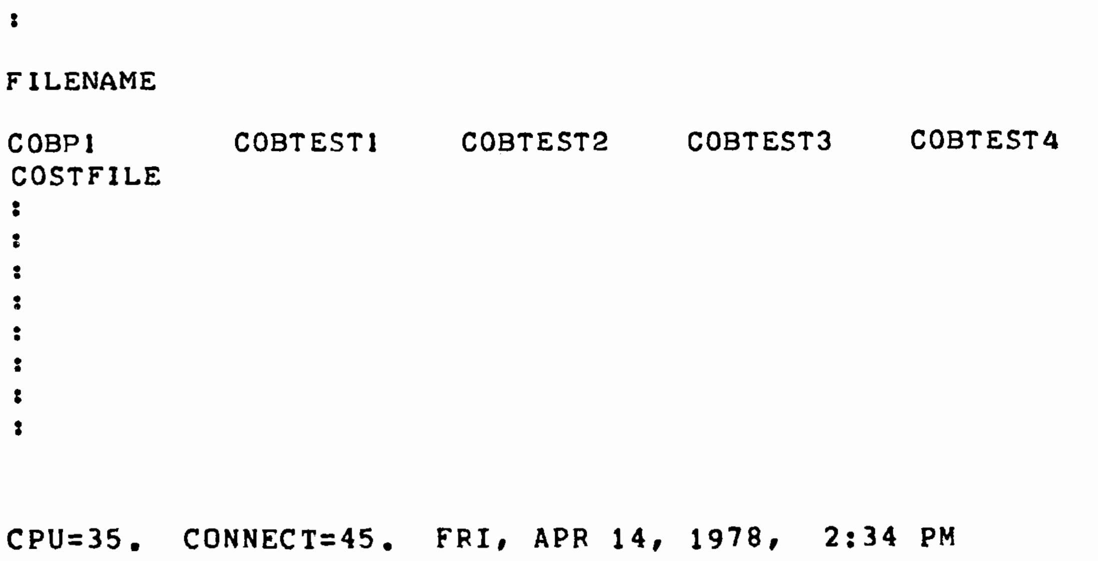
```
:

PAGE 0001   HP32213C.02.00  (C) HEWLETT-PACKARD CO. 1977


     ØØ11ØØ IDENTIFICATION DIVISION.
     ØØ12ØØ PROGRAM-ID. COBOL-TEST4.
     ØØ13ØØ AUTHOR. YOUR NAME.
     ØØ14ØØ ENVIRONMENT DIVISION.
     ØØ141Ø INPUT-OUTPUT SECTION.
     ØØ142Ø FILE-CONTROL.
     ØØ143Ø SELECT COST-FILE ASSIGN TO "COSTFILE".
     ØØ15ØØ DATA DIVISION.
     ØØ151Ø FILE SECTION.
     ØØ152Ø FD  COST-FILE
     ØØ153Ø         LABEL RECORD IS OMITTED.
     ØØ154Ø Ø1  COST-REC.
     ØØ155Ø     Ø5  DATE-FIELD     PIC X(8).
     ØØ156Ø     Ø5  T-COST-FIELD   PIC 9999V99.
     ØØ16ØØ WORKING-STORAGE SECTION.
     ØØ17ØØ 77  EDIT-FIELD      PIC $Z,ZZ9.99.
     ØØ18ØØ 77  TOTAL-COST      PIC 999V99.
     ØØ2ØØØ

     ØØ21ØØ PROCEDURE DIVISION.
     ØØ22ØØ OPEN-ROUTINE.
     ØØ23ØØ     OPEN INPUT COST-FILE.
     ØØ24ØØ     MOVE ZEROS TO TOTAL-COST.
     ØØ25ØØ READ-ROUTINE.
     ØØ26ØØ     READ COST-FILE AT END GO TO GRAND-TOT-ROUTINE.
     ØØ27ØØ     ADD T-COST-FIELD TO TOTAL-COST.
     ØØ28ØØ     MOVE T-COST-FIELD TO EDIT-FIELD
     ØØ29ØØ     DISPLAY "TRANSACTION DATE = " DATE-FIELD
     ØØ3ØØØ             "  TOTAL COST = " EDIT-FIELD.
```



```
  ØØ31ØØ      GO TO READ-ROUTINE.
  ØØ32ØØ  GRAND-TOT-ROUTINE.
  ØØ33ØØ      MOVE TOTAL-COST TO EDIT-FIELD.
  ØØ34ØØ      DISPLAY SPACE.
  ØØ35ØØ      DISPLAY "                    GRAND TOTAL TO DATE ="
  ØØ36ØØ              EDIT-FIELD.
  ØØ37ØØ      CLOSE COST-FILE.
  ØØ38ØØ      STOP RUN.

    DATA AREA IS %ØØØ367 WORDS.
    CPU TIME = Ø:ØØ:Ø5.  WALL TIME = Ø:Ø3:17.
END COBOL/3ØØØ COMPILATION.  NO ERRORS.  NO WARNINGS.

 END OF COMPILE

 END OF PREPARE

TRANSACTION DATE = Ø5/12/75  TOTAL COST = $   13.Ø8
TRANSACTION DATE = Ø5/12/75  TOTAL COST = $    1.Ø6
TRANSACTION DATE = Ø5/12/75  TOTAL COST = $   1Ø.6Ø

                    GRAND TOTAL TO DATE = $   24.74

 END OF PROGRAM
:
```

The three "total cost" figures were read out of the COSTFILE. The "grand total" figure was computed by this program.

This completes the demonstrations for this section. Purge all unneeded files and, if you are done for this session, log off. (If necessary, use LISTF to see which files actually exist in your file group.)



```
:

FILENAME

COBP1        COBTEST1     COBTEST2     COBTEST3     COBTEST4
COSTFILE
:
:
:
:
:
:
:
:


CPU=35.  CONNECT=45.  FRI, APR 14, 1978,  2:34 PM
```

A final note: The *Using COBOL* guide (Part number 32213-90003) contains a great amount of information about COBOL/3000 as it relates to MPE (the HP 3000 operating system), as well as specific information about the use of EDIT/3000, the segmenter, and other advanced COBOL/3000 features. It should prove extremely useful to you as a new COBOL/3000 user.

summary

:BUILD COSTFILE;REC=-14,18,F,ASCII

builds an empty system file of 1023 records, 14 characters per record, 18 records per block

1.4 SELECT COST-FILE ASSIGN TO "COSTFILE".

assigns COBOL file name COST-FILE to system file name COSTFILE

:FILE COSTFILE;ACC=APPEND

allows adding more data to COSTFILE in next run in this session

:FILE COSTFILE;DEV=LP

specifies COSTFILE as line printer

:RESET COSTFILE

nullifies any previous :FILE commands for COSTFILE

:LISTF COSTFILE,1

displays number of used records, their size and type, and total records allowed for COSTFILE

section 6
using BASIC/3000

summary of commands introduced in this section

>SAVE BAS2!,FAST	! purges old file of same name before saving
>CREATE DFILE,10	creates empty 10-record BASIC formatted data file
>EXIT	exits the BASIC/3000 subsystem
10 FILES DFILE,ASCF	assigns DFILE as file #1 and ASCF as file #2 in this program
20 PRINT #1;A	serial write
30 PRINT #1,5;A	direct write
40 READ #1?A	serial read
50 READ #1,5;A	direct read
60 RESTORE #1	positions file #1 pointer to beginning of file
70 READ #2,2	positions file #2 pointer to beginning of record 2
80 ADVANCE #1;10,X	advances file #1 pointer by ten items
90 LINPUT #2,3;B$	reads entire record 3 of file 2 into B$
100 B$=A$(3)	substring is character 3 to end of A$
110B$=A$(3,5)	substring starts at character 3, ends at character 5 of A$
120B$=A$(3;6)	substring starts at character 3, six characters long
>SAVE BAS1	names and saves work area as program file BAS1
>SAVE BAS2,FAST	same except saved program is faster to get, chain, invoke
>RUN BAS1	gets and runs BAS1
10 CONVERT A TO A$	converts value to string
20 CONVERT A$ TO A	converts string to value
30 B=RND(0)	generates random number ($0 \leqslant B < 1$)
40 INPUT "AGE?",A	input prompt string replaces question mark prompt
50C$=D$+"LESS"	+ operator concatenates two strings
>SYSTEM :FILE PRINTER;DEV=LP :RESUME	makes line printer available for listings and output
>LIST,OUT=PRINTER	lists work area on line printer
>RUN,OUT=PRINTER	prints program output on line printer

using BASIC/3000

BASIC/3000 is an interpreter language which includes all the familiar commands and statements common to most forms of BASIC (LET, READ, PRINT, GOTO, IF-THEN, FOR-NEXT-STEP, SAVE, GET, LIST, RUN, etc.). We assume you are familiar with at least these fundamental operations in the BASIC language.

BASIC/3000, however, also includes many extensions found only in BASIC/3000. We will cover a few of these extensions in this section, but by no means all of them. To take full advantage of everything that is available to you, be sure to consult the BASIC/3000 Interpreter reference manual or pocket guide.

differences from other BASIC interpreters

Many things you are used to are still the same. You number your statements by adequate intervals (e.g., 10) to allow for line insertions. You replace a line by retyping with the same line number. Comma separators in a PRINT list cause items to be spaced out in five fields per line; semicolons pack items as tightly as the lengths of the items allow. And your work area disappears when you terminate use of the Interpreter.

But also note the following items. Some of these may be different from what you're used to.

Log-On and Typing Errors. Be sure you have read Section 1. At this point we assume you know how to log on, how to delete typed-in characters, and how to log off.

Prompt Character. The prompt character for BASIC/3000 is >. Other systems may use a different character or no prompt at all. After the Interpreter prints out the >, you may enter any valid BASIC/3000 statement or command.

Syntax Errors. When the Interpreter detects an error in a statement typed in (such as in the example below), an error message will be printed out on the next line:

```
>1Ø PRINT 5*SIX
ERROR@9
```

This means that nine non-blank characters were successfully processed before the error was detected. You can either press RETURN to retype immediately or, to get an expanded explanation of the error, type any printing character (except a colon); for example, a dash:

```
ERROR@9-
ILLEGAL ITEM IN NUMERIC EXPRESSION
>
```

Numeric Type. All simple and array variables are assumed to be type REAL unless specifically declared otherwise (integer, long, or complex). Details on numeric type conversion are given in the reference manual.

Line Numbers. The range is 1 to 9999.

Purging Files. Program and data files are eliminated from the system by a PURGE statement. This is in order to be compatible with the :PURGE command of the MPE/3000 operating system. Other systems may use such words as KILL or UNSAVE.

Breaks. To terminate endless loops or commands that are taking too long to execute, use Y^c. Then (in the case of the suspended program) type ABORT after the > prompt.

There is also a different kind of break, using the SYSTEM command, which temporarily suspends your use of the Interpreter so you can do something in system mode. To return to BASIC from this kind of break, type RESUME after a colon prompt.

Operators. Use:

**	for exponentiation
<>	for does-not-equal
<=	for is-less-than-or-equal-to
>=	for is-greater-than-or-equal-to

entering and running a program

If you are not currently logged on, log on now. Then access the BASIC/3000 Interpreter by typing BASIC in response to a colon prompt. The system will then load the Interpreter from disc if no one else is currently using it; this may take a few seconds.

```
:       ED.DATASYS3
HP3000 / MPE III.  TUE, MAR 7, 1978,  10:50 AM
:
BASIC 3.Ø
>
```

When you have access to the Interpreter, it prints out its name and version number. This is followed by the first > prompt. You may then begin entering your program. The program shown below is a simple one which should contain no surprises. Its intent is to read an input value from the terminal, read a value from a DATA statement, do a simple computation, and print out the result. This is done three times, after which the program terminates.

```
>
>
>
>
>
>
>
>
>
>
>
>
>
>
```

Now name the program and save a permanent copy in your library.

```
>
>
>
```

Another way to do this is with only one command: SAVE BAS1. This both names the copy and saves it. In this case, the copy remaining in the work area remains,unnamed.

You can check what files you have saved with a CAT (abbreviated form of CATALOG) command.

```
>


ACCOUNT=DATASYS3   GROUP=PUB

   NAME     RECORDS    NAME     RECORDS
BAS1       SP      4
>
```

This tells you that your program uses four records (A record in a program file normally contains 128 words); also that the file is a "saved program" (SP). Other file types are: fast-saved program (FP), BASIC data file (BF), ASCII file (A), and binary file (B).

You can use GET BAS1 and then RUN, just as with most interpreters. But, in addition, BASIC/3000 will allow you to use RUN BAS1, which will both get and run the program. Try it, and, (enter four values) when asked for by the program, (one deliberately too large).

```
>
BAS1
ENTER VALUE, LESS THAN 1ØØ
?
ANSWER IS:        .6Ø
ENTER VALUE, LESS THAN 1ØØ
?
BAD VALUE. TRY AGAIN.
?
ANSWER IS:        .38
ENTER VALUE, LESS THAN 1ØØ
?
ANSWER IS:      -.17
>
```

List the program. Note that FOR-loops are indented whether or not you did so originally. Nested FOR-loops would be indented further.

```
>
BAS1
  1Ø FOR I=1 TO 3
  2Ø   PRINT "ENTER VALUE, LESS THAN 1ØØ"
  3Ø   INPUT A
  4Ø   IF A>=1ØØ THEN 1ØØ
  5Ø   READ B
  6Ø   C=SIN(A)+COS(B)
  7Ø   PRINT USING 12Ø;C
  8Ø NEXT I
  9Ø STOP
 1ØØ PRINT "BAD VALUE. TRY AGAIN."
 11Ø GOTO 3Ø
 12Ø IMAGE "ANSWER IS:",6D.DD
 13Ø DATA 33.3,5Ø,66.7
>
```

selected extensions of the language

Many of the extensions to the BASIC language provided by BASIC/3000 are intended for advanced applications. Others, such as those used in the example shown on the next page, are not difficult to use. These have been chosen as being particularly useful to the beginner as well as the advanced user.

As indicated by the notes adjacent to the two example programs, the various statements and commands demonstrate the use of:

- common data
- substrings
- default strings
- number-to string conversion
- input prompt strings
- DO clause
- invocation of programs
- the IMAGE statement
- format strings
- the continuator
- the concatenate operator
- compressed (K) format
- the save-fast option
- output FOR-loops

The two programs in the example are named BAS2 and ALLIST. BAS2 generates ten random phone numbers, and allows a user to choose to display one of the ten at a time, or to display all of them at once, and then stop. BAS2 displays all of the phone numbers by invoking ALLIST, which returns the phone numbers under the heading "THE NUMBERS ARE:".

BAS2 will invoke ALLIST if the user responds "0" to the input prompt (see lines 100 and 110); after being invoked, ALLIST returns control to BAS2 at line 120. Execution then stops at line 130. Alternatively, if the user does not input zero, only a single element is printed, and if the user answers "N" in line 160, then execution stops. The END statement at the end of ALLIST is not required; it is included in this example to show the contrast between it and the STOP statement. If STOP were used in place of END, execution of both programs would terminate at that point and there would be no return to BAS2.

Now let's take a closer look at the programs. Line 10 in the first program sets up a block of data storage common to BAS2 and ALLIST; it is to contain a string array having ten elements of eight characters each. Lines 20 through 80 fill the array with ten random "phone numbers." This is done by converting two random numbers to their corresponding ASCII equivalents, one three digits long, and the other four digits long, and then joining the two sets of digits with a hyphen between them. This is repeated ten times, storing each phone number in successive elements of the string array C$. (More details: RND generates a value between 0 and 1; multiplying by 10^6 removes the decimal point. Line 50 examines the first character of the result and rejects the entire number if the first character is 0; this is done because phone numbers don't normally begin with 0.)

The quoted prompt string in line 90 replaces the normal question mark prompt for the INPUT statement. The response is input to A. If A is 0, lines 100 through 140 are executed; if A is non-zero, execution branches to line 150.

In the first case the DO clause immediately invokes the program ALLIST. ALLIST picks up the array of phone numbers through passed common and prints out each number by means of an output FOR-loop. The FOR-loop is an alternative to the MAT PRINT statement and has the advantage of permitting selected portions of the array to be printed out. In this case, the entire array is printed. (INPUT and READ also permit this kind of FOR-loop.) The format used here is expressed in a quoted format string: four spaces followed by eight ASCII characters, occurring twice per line, then followed by a carriage return-line feed(/). This pattern is repeated until all ten elements have been printed. Upon return to BAS2, line 120 is executed. This prints out the image in line 180.

BAS2

```
>SCR
>1Ø COM C$(1Ø,8)                                     Common array (string)
>2Ø DIM A$(3),B$(4)
>3Ø FOR N=1 TO 1Ø
>4Ø   CONVERT RND(Ø)*1Ø**6 TO A$                     Conversion to string
>5Ø   IF A$(1,1)="Ø" THEN GOTO 4Ø                    Substring
>6Ø   CONVERT RND(Ø)*1Ø**6 TO B$
>7Ø   C$(N)=A$+"-"+B$                                String concatenation
>8Ø NEXT N
>9Ø INPUT "WHICH ARRAY ELEMENT (1 TO 1Ø)? ",A        Input prompt string
>1ØØ IF A=Ø THEN DO                                  DO clause
>11Ø   INVOKE "ALLIST"                               Invoking another program
>12Ø   PRINT USING 18Ø;N-1                           IMAGE; preservation of variables after INVOKE
>13Ø   STOP
>14Ø DOEND
>15Ø PRINT USING "17A,8A/";"PHONE NUMBER "+&         Format string; continuator
>>"IS: ",C$(A)                                       Continuation prompt
>16Ø INPUT "ANY OTHERS? Y OR N. ",D$                 Default string
>17Ø IF D$="Y" THEN GOTO 9Ø
>18Ø IMAGE "ALL",XKX,"NUMBERS LISTED."               Compressed (K) format
>SAVE BAS2,FAST                                      Save-fast option
```

ALLIST

```
>SCR
>1Ø REM
>2Ø REM      COMPLETE PHONE LIST PROGRAM
>3Ø REM
>4Ø COM G$(1Ø,8)                                     Passed common
>5Ø PRINT "THE NUMBERS ARE:"
>6Ø PRINT USING "2(4X,8A)/";(FOR K=1 TO 1Ø,G$(K))    Output FOR-loop
>7Ø END
>SAVE ALLIST,FAST
>
```

The value N-1 replaces K in the IMAGE statement (preceded and followed by one blank), resulting in the message "All 10 numbers listed." being printed. The compressed (K) format allows numbers to be inserted in text without leading and trailing blanks. Note that N, which has been incremented to 11 on the last execution of line 80, is still valid on return from ALLIST, since INVOKE preserves existing variables. After the message is printed, the program terminates at line 130.

If the response in line 90 is non-zero, the branch to line 150 causes a 17-character string ("Phone number is:") to be printed out, followed by the selected element from array C$. Only the numbers 1 through 10 are valid. (Note the use of the & continuator and the double >> prompt on the next line.) After this program asks "Any others?", to which the user responds Y (Yes) to continue, or anything else to terminate. Note that D$ is not dimensioned in the program; it is a "default" string, one character in length.

Now let's try the programs.

Enter and save the two programs exactly as shown above. Then type RUN BAS2. The program then runs and prints out its first prompt.

```
>RUN BAS2
BAS2
WHICH ARRAY ELEMENT (1 TO 1Ø)?
```

Respond with a whole number from 1 to 10, and answer "Y" when asked "Any others?" Try another value and another "Y". Then enter a 0 to get the complete list of ten numbers.

```
WHICH ARRAY ELEMENT (1 TO 1Ø)? 3
PHONE NUMBER IS: 5Ø3-121Ø

ANY OTHERS? Y OR N. Y
WHICH ARRAY ELEMENT (1 TO 1Ø)? 1Ø
PHONE NUMBER IS: 75Ø-3832

ANY OTHERS? Y OR N. Y
WHICH ARRAY ELEMENT (1 TO 1Ø)? Ø
THE NUMBERS ARE:
    561-5869     286-9528
    5Ø3-121Ø     734-9152
    617-5Ø42     146-3714
    2Ø2-2423     675-1Ø77
    959-9658     75Ø-3832

ALL 1Ø NUMBERS LISTED.

>
```

The list of the ten numbers and the heading "The numbers are:" were printed out by the program ALLIST. Everything else was done by the program BAS2. The ten numbers were passed to the ALLIST program through common storage. In the next example the numbers are "passed" through a file.

summary

>SAVE BAS1	names and saves work area as program file BAS1
>SAVE BAS2,FAST	same except saved program is faster to get, chain, invoke
>RUN BAS1	gets and runs BAS1
10 CONVERT A TO A$	converts value to string
20 CONVERT A$ TO A	converts string to value
30 B=RND(0)	generates random number ($0 \leqslant B < 1$)
40 INPUT "AGE?",A	input prompt string replaces question mark prompt
50 C$=D$+"LESS"	+ operator concatenates two strings.

using files in BASIC

BASIC/3000 permits the use of three types of files: BASIC formatted files, ASCII files, and binary files. The first of these is created under control of the Interpreter, using the Interpreter's CREATE command or statement. Special formatting allows the Interpreter to check for possible data type mismatch during the execution of your programs. The other files are usually created externally, such as by using MPE/3000's BUILD command or EDIT/3000's KEEP command.

In this section we use a BASIC formatted file and an ASCII file. The ASCII file is prepared using the Editor subsystem as an example; such a file could as well be prepared through use of any of the HP 3000 languages. Special provisions (direct access method and the LINPUT statement) allow easy access to record-oriented ASCII data.

BASIC FORMATTED FILES

BASIC formatted files are easy to use within BASIC/3000 and have built-in error checking features to help protect file integrity. A particular advantage is that these files permit use of the DUMP command and ADVANCE statement, which provide simple means of examining the contents of data files and accessing individual items in the file.

The diagram on this page shows the two methods of file access: serial access and direct access. Both cases assume a three record file; that is, they could have been created with a command or statement like: CREATE SERFILE,3.

Serial Access. Using serial access permits you to ignore record boundaries. For example if, in printing an array to a file, the array is larger than the record size (106 words is the default case), the Interpreter simply continues writing into the next record. A succeeding print statement to the same file would continue writing data following the last word of the array.

This method allows you to work strictly with "items." You need not be concerned with words, records, or sectors. For example, if you want to access the 120th item in the file, use RESTORE to reset the file pointer to the beginning of the file, ADVANCE 119 to skip to the desired item, and READ to read out the item.

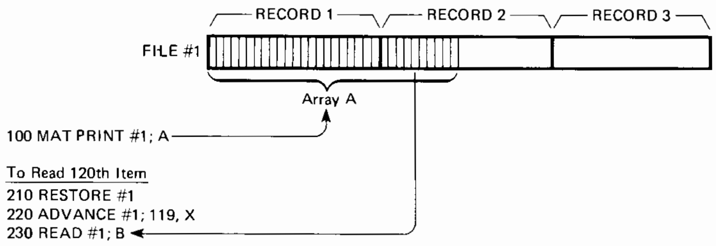
Direct Access. Using direct access requires planning and structuring of your data to fit the record size. Also, when accessing records, you should always be aware of the logical structure of your records; that is, the number of items per record and the order in which they are stored.

In planning your data structure for direct access you need to know the following facts:

1. The default logical record size is 106 words
2. Numbers use:

   1 word for integers
   2 words for type real
   4 words for type long
   4 words for type complex

SERIAL ACCESS

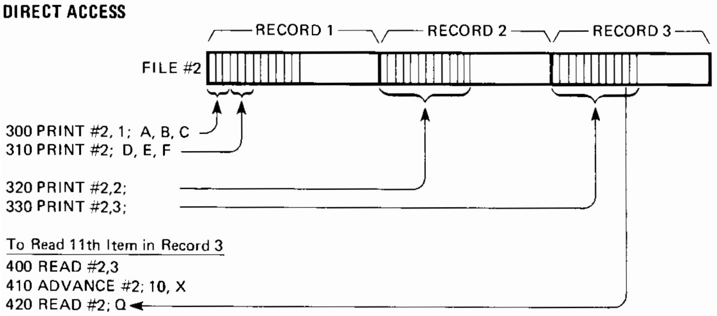


```
100 MAT PRINT #1; A
```

To Read 120th Item

```
210 RESTORE #1
220 ADVANCE #1; 119, X
230 READ #1; B
```

DIRECT ACCESS



```
300 PRINT #2, 1;  A, B, C
310 PRINT #2;  D, E, F

320 PRINT #2,2;
330 PRINT #2,3;
```

To Read 11th Item in Record 3

```
400 READ #2,3
410 ADVANCE #2; 10, X
420 READ #2; Q
```

3. Strings use half a word (one byte) for each character, plus one word for the Interpreter to use for a "length word." Thus a 12-character string uses 7 words (12/2 + 1). If a string has an odd number of characters, it requires a full word to contain the last character. Thus, a 13-character string uses eight words (12/2+1+1).

Direct access is particularly efficient for structured data. As shown in the diagram, you can write data to any desired record with a statement such as

```
PRINT #2,3,;A,B,C
```

Which would write A, B, and C into record 3 of file 2. To read out a data item, you can also go directly to the record of interest. First use READ # without parameters (e.g., READ #2,3) to position the file #2 pointer to the beginning of the record (record 3). Then use ADVANCE to move the pointer ahead to the desired item and do a "serial" read. (A direct read such as READ #2,3;Q would force the pointer back to the beginning of record 3.)

Example Programs. To demonstrate the use of BASIC formatted files, we'll make a few modifications to the BAS2 and ALLIST programs used in the previous exercise. These modifications eliminate the common storage previously used to pass the ten phone numbers to ALLIST. Instead, a BASIC formatted file is created into which BAS2 writes the ten phone numbers. ALLIST then accesses the file to list them out when requested.

Do the following.

```
>
>
>
>
>
>
>
>
>
>
>
>
```

The CREATE command creates a 10-record file for you. The new statement 10 assigns FONELIST the file number of #1 since it is the first (and only) file name listed by a FILES statement. The new statement 70 writes the fabricated phone numbers into file #1 (instead of into the former string array); and the new statements 20, 145, 147, 150, and 155 arrange serial access of the items in file #1. ADVANCE specifies A–1 rather than A since the file pointer is already pointing at item 1 before execution begins; the parameter X is not used here, but should always be checked in more complex programs (0 if advance was successful).

The complete program can now be listed.

```
>
BAS2
  1Ø FILES FONELIST
  2Ø DIM A$[3],B$[4],C$[8]
  3Ø FOR N=1 TO 1Ø
  4Ø   CONVERT RND(Ø)*1Ø**6 TO A$
  5Ø   IF A$[1,1]="Ø" THEN GOTO 4Ø
  6Ø   CONVERT RND(Ø)*1Ø**6 TO B$
  7Ø   PRINT #1;A$+"-"+B$,END
  8Ø NEXT N
  9Ø INPUT "WHICH ARRAY ELEMENT (1 TO 1Ø)? ",A
 1ØØ IF A=Ø THEN DO
 11Ø   INVOKE "ALLIST"
 12Ø   PRINT USING 18Ø;N-1
 13Ø   STOP
 14Ø DOEND
 145 RESTORE #1
 147 ADVANCE #1;A-1,X
 15Ø READ #1;C$
 155 PRINT USING "17A,8A/";"PHONE NUMBER "+"IS: ",C$
 16Ø INPUT "ANY OTHERS? Y OR N. ",D$
 17Ø IF D$="Y" THEN GOTO 9Ø
 18Ø IMAGE "ALL",XKX,"NUMBERS LISTED."
>
```

This being satisfactory, we can save the program as the new BAS2. An exclamation mark after the file name in a SAVE command automatically purges the old BAS2. It saves having to enter a separate PURGE command.

```
> [illegible]
>
```

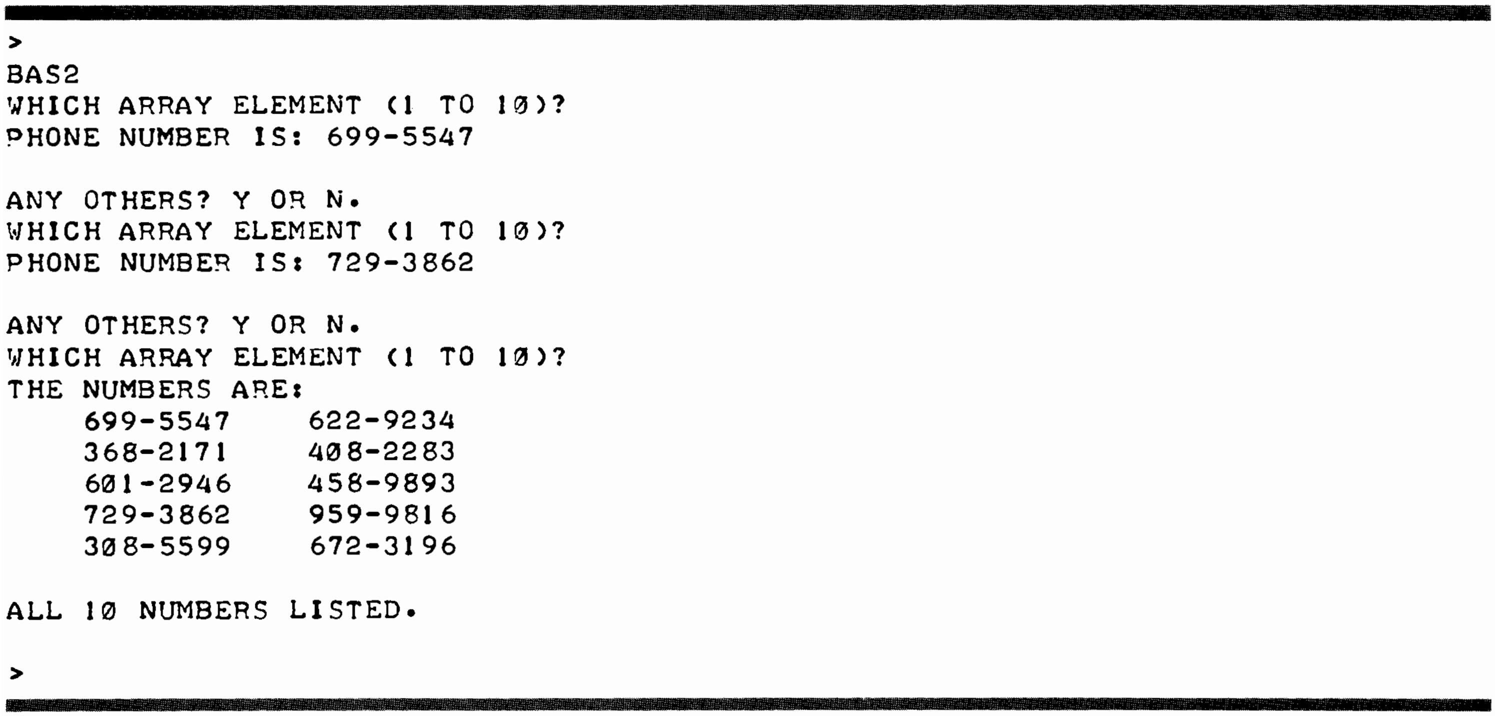
Now we have to modify ALLIST so that it is able to access the FONELIST file.

```
> [illegible]
> [illegible]
> [illegible]
> [illegible]
```

The new statement 40 dimensions a string array to receive the list of phone numbers. Statement 45 declares FONELIST to be file #1 in this program, and statement 55 reads the entire file contents into the string array G$. Use LIST to check the program, and save it using the purge option.

```
>LIST
ALLIST
  10  REM
  20  REM      COMPLETE PHONE LIST PROGRAM
  30  REM
  40  DIM G$[10,8]
  45  FILES FONELIST
  50  PRINT "THE NUMBERS ARE:"
  55  MAT READ #1;G$
  60  PRINT USING "2(4X,8A)/";(FOR K=1 TO 10,G$[K])
  70  END
>
>
```

Now we can run the programs in the same manner as before. There is no visible difference, but you have used a BASIC formatted file.

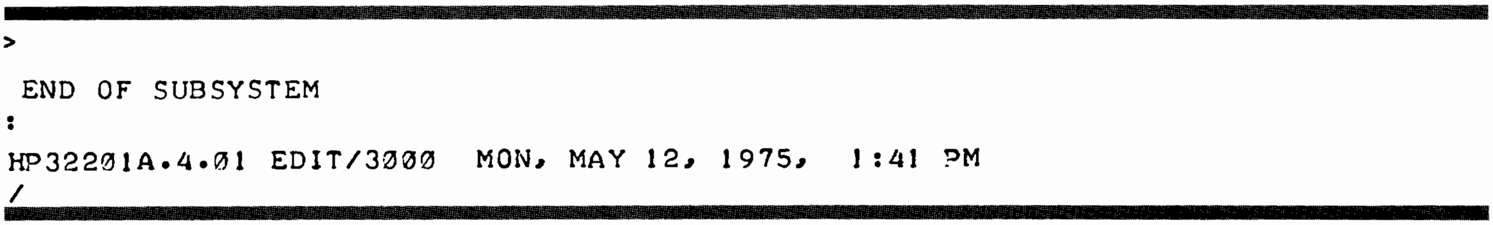


```
>
BAS2
WHICH ARRAY ELEMENT (1 TO 1Ø)?
PHONE NUMBER IS: 699-5547

ANY OTHERS? Y OR N.
WHICH ARRAY ELEMENT (1 TO 1Ø)?
PHONE NUMBER IS: 729-3862

ANY OTHERS? Y OR N.
WHICH ARRAY ELEMENT (1 TO 1Ø)?
THE NUMBERS ARE:
    699-5547     622-9234
    368-2171     4Ø8-2283
    6Ø1-2946     458-9893
    729-3862     959-9816
    3Ø8-5599     672-3196

ALL 1Ø NUMBERS LISTED.

>
```

ASCII FILES

To illustrate the use of ASCII files we will use the EDIT/3000 sybsystem to create a file containing ten "employee names." We assume they are entered in the file in the same order as the ten phone numbers presently existing in the FONELIST file. Thus we can write a simple program that will match names to phone numbers.

Begin by exiting from the BASIC/3000 subsystem and accessing the EDIT/3000 subsystem. To exit, simply type EXIT in response to the BASIC/3000 prompt.



```
>

 END OF SUBSYSTEM
:
HP322Ø1A.4.Ø1 EDIT/3ØØØ  MON, MAY 12, 1975,  1:41 PM
/
```

Now type in an "ADD" command and wait for the carriage or cursor to stop moving after printing out the line "1" prompt. Then begin entering employee numbers and names exactly as shown below. That is: four digits, one space, one initial, one space, and surname.

```
/
      1
      2
      3
      4
      5
      6
      7
      8
      9
     1Ø
     11
```

To terminate the ADD command press control-Y (Y^{c}). Three dots are then printed out followed by a new slash prompt.

```
     11        ...
/
```

Now type KEEP ASCFILE. This causes the Editor to create an ASCII file (with fixed-length 80-character records) having the name ASCFILE. The ten lines you just typed in are stored as ten records in that file.

```
/
/
```

To return to BASIC, type END, press the RETURN key, and type BASIC in response to the colon prompt.

```
/

 END OF SUBSYSTEM
:

BASIC 3.Ø
>
```

Now enter the following program and save it as BAS3.

```
>10   FILES FONELIST,ASCFILE
>[illegible]
>[illegible]
>[illegible]
>[illegible]
>60      [illegible] #1
>70      [illegible]
>[illegible]
>[illegible]
>[illegible]
>110        [illegible] C$,B$(6),B$[illegible]
>120        IMAGE [illegible]
>130     NEXT N
>140     STOP
>150 DOEND
>160 FOR N=1 TO 10
>170       INPUT #2,[illegible]
>[illegible]
>[illegible]
>[illegible]
>[illegible]
>220        [illegible]
>230        GOTO [illegible]
>240     DOEND
>250 NEXT N
>[illegible] INPUT "***[illegible]
>[illegible]
>
>
```

Line 10 assigns FONELIST as file #1, and ASCFILE as file #2. If "ALL" is input for line 30, the FOR-loop in lines 80 to 130 is executed ten times. Line 90 successively inputs each entire record of ASCFILE into the string B$; this is direct access. Then line 100 successively reads each phone number from FONELIST into the string C$; this is serial access. Then, knowing the structure of the data, we can print out all information by the statements in lines 110 and 120. That is, knowing that the employees' names begin at character 6 in B$, we can print B$(6) (which extends from character 6 to the logical end of the string) in a 12-character ASCII field. Then, following "EMPL#", the employee number (which is in characters 1 through 4 of B$) is printed in a 4-digit ASCII field. Lastly, the 8-character phone number is printed.

If anything but NONE or ALL is input for line 30, the FOR-loop in lines 160 to 250 is executed until a match is found for the input string. Line 170 successively inputs each entire record of ASCFILE into the string B$, just like line 90. Line 180 compares the input string (A$) with the substring of B$ which begins with character 8 (where the surname starts) and which has the same number of characters (length) as A$. If no match is found, the FOR-loop repeats. When a match is found, the Nth phone number is read from FONELIST into C$ (lines 190, 200, 210) and is printed out according to line 220. Line 230 then returns control to line 30. If no match is found after 10 loops, lines 260 and 270 are executed.

Now run the program. Enter a surname from the list you put into ASCFILE earlier. Then try another. Then enter ALL. Your display should look like this:

```
>
NAME?
PHONE NUMBER IS: 368-2171
NAME?
PHONE NUMBER IS: 959-9816
NAME?
THE NUMBERS ARE:
   E ALBERT     (EMPL# Ø843):   699-5547
   J BELL       (EMPL# Ø576):   622-9234
   P CARR       (EMPL# ØØ92):   368-2171
   Q DEXTER     (EMPL# 1438):   4Ø8-2283
   R EVANS      (EMPL# 1755):   6Ø1-2946
   G FAY        (EMPL# Ø948):   458-9893
   S GOLD       (EMPL# Ø663):   729-3862
   B HILL       (EMPL# 1349):   959-9816
   O HOWE       (EMPL# Ø252):   3Ø8-5599
   A IDE        (EMPL# 1124):   672-3196
```

All the phone numbers you see displayed came from the BASIC formatted file FONELIST. All the names and employee numbers came from the ASCII file ASCFILE.

As you become more familiar with using ASCII files in BASIC, the following techniques will prove extremely useful.

a. You can convert a BASIC formatted file to an ASCII file (perhaps for use in another language) by the following method. First build an empty ASCII file by temporarily going to system mode:

```
>
```

Then use READ# and PRINT# statements to copy each record from the BASIC file to the ASCII file.

b. If you want to create an ASCII file in BASIC that can be read back in BASIC, you must remember that READ# expects commas between data items and quotes around strings. PRINT# does not do this. You can insert your own commas and quotes like this:

The '34 inserts ASCII quote characters around A$, and the quoted comma inserts a comma between the two items.

c. After reading an ASCII record into a BASIC string (perhaps with LINPUT#) you can convert any ASCII numerical characters to the equivalent numerical value using CONVERT. For example, if you know that characters 40 through 49 are a number you want to work with, you can use:

The number X is then usable for arithmetic operations.

summary

>SAVE BAS2!,FAST	! purges old file of same name before saving
>CREATE DFILE,10	creates empty 10-record BASIC formatted data file
>EXIT	exits the BASIC/3000 subsystem
10 FILES DFILE,ASCF	assigns DFILE as file #1 and ASCF as file #2 in this program
20 PRINT #1;A	serial write
30 PRINT #1,5;A	direct write
40 READ #1;A	serial read
50 READ #1,5;A	direct read
60 RESTORE #1	positions file #1 pointer to beginning of file
70 READ #2,2	positions file #2 pointer to beginning of record 2
80 ADVANCE #1;10,X	advances file #1 pointer by ten items
90 LINPUT #2,3;B$	reads entire record 3 of file 2 into B$
100 B$=A$(3)	substring is character 3 to end of A$
110 B$=A$(3,5)	substring starts at character 3, ends at character 5 of A$
120 B$=A$(3;6)	substring starts at character 3, six characters long

using the line printer

You can direct the hard copy for program listings or for program output to the system line printer by using an OUT=parameter in the LIST or RUN command.

Before you proceed, however, inquire of someone in your area who is familiar with the use of your particular system's line printer. Administration differs in various installations. The device name is commonly "LP", but this may not be true of your system — in fact, you may have more than one line printer available. Also, you may have to make special arrangements with the system operator to retrieve your line printer copy. When you are ready, proceed as follows.

Use the SYSTEM command to go temporarily into the system mode, and enter a :FILE equation.

```
>


:
:
```

This states that the device LP will be referenced each time you use the formal file designator PRINTER in your program. (This method could also be used for other unit record devices, such as card punches and magnetic tape units.)

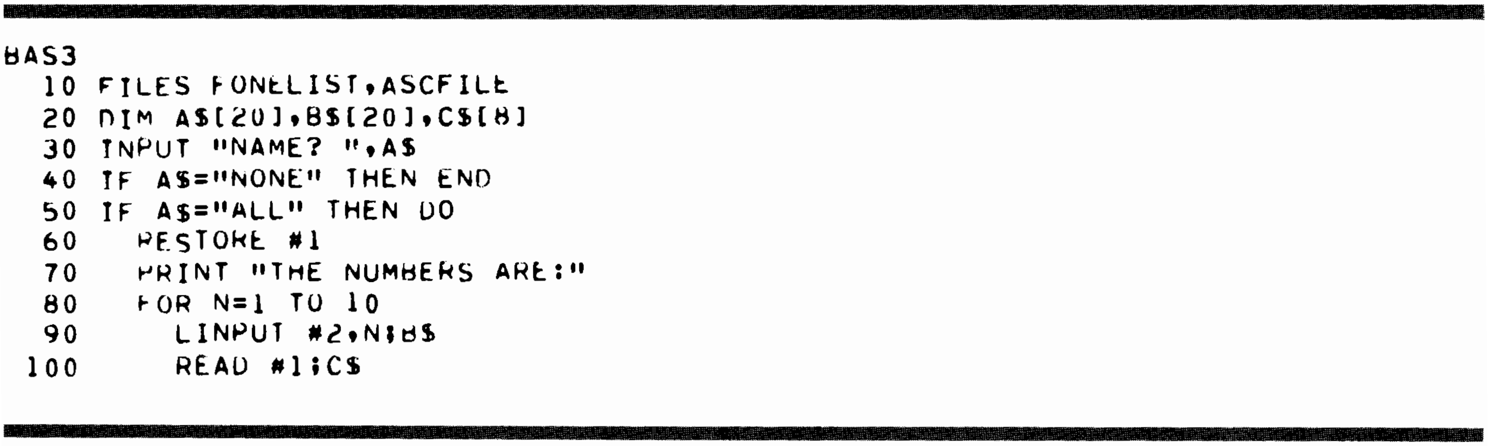
Now return to BASIC with RESUME. Then list your program on the line printer by using OUT=PRINTER.

```
:
>
>
```

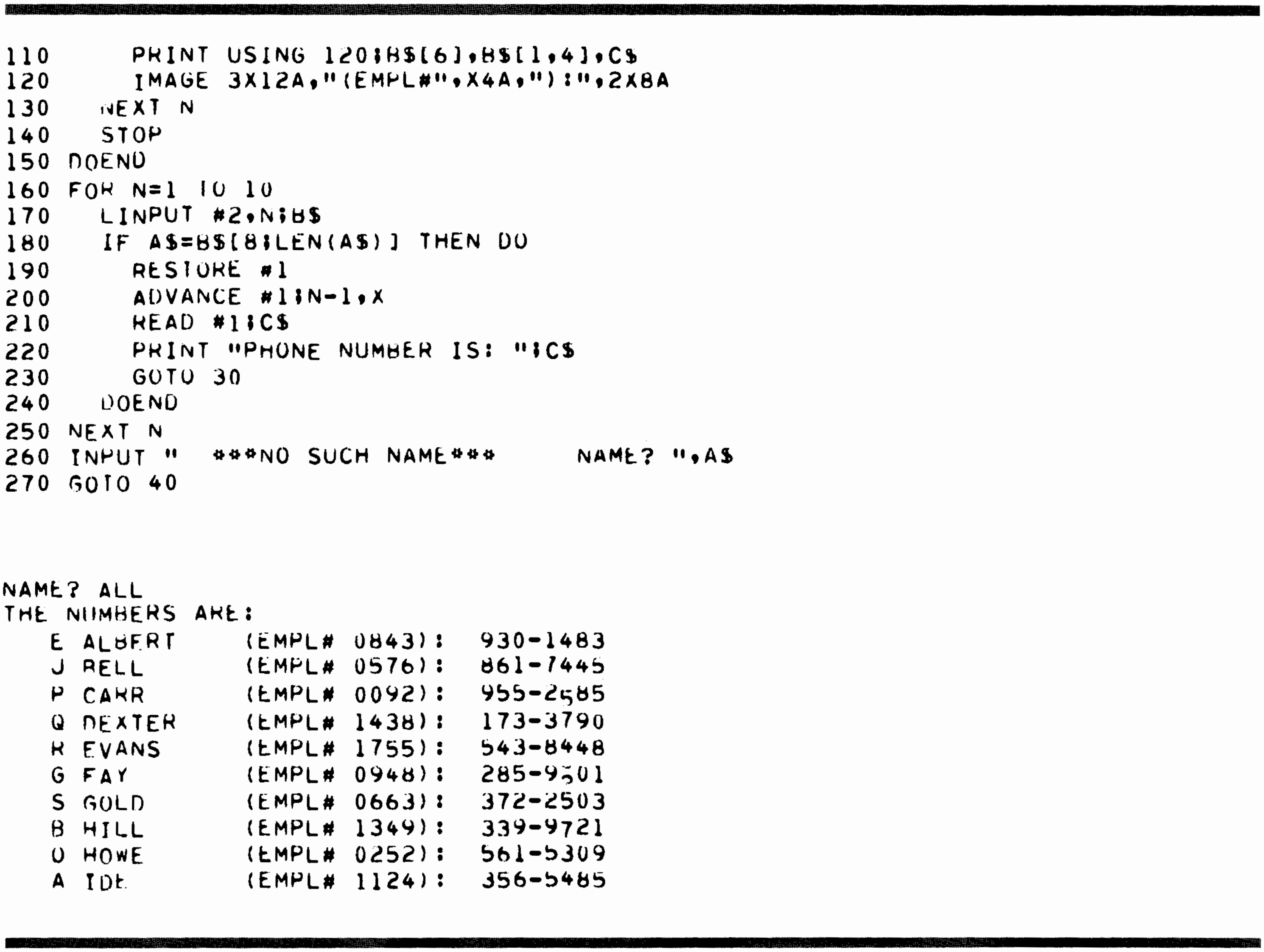
Run the program with output directed to the line printer. (Note that input prompts are duplicated at the terminal.) Lastly, exit from BASIC and retrieve your line printer copy.

```
>
NAME?
>

 END OF SUBSYSTEM
```

Your copy should look like the following sample (which here has been cut and joined from two separate pages).



```
BAS3
  10 FILES FONELIST,ASCFILE
  20 DIM A$[20],B$[20],C$[8]
  30 INPUT "NAME? ",A$
  40 IF A$="NONE" THEN END
  50 IF A$="ALL" THEN DO
  60   RESTORE #1
  70   PRINT "THE NUMBERS ARE:"
  80   FOR N=1 TO 10
  90     LINPUT #2,N;B$
 100     READ #1;C$
```

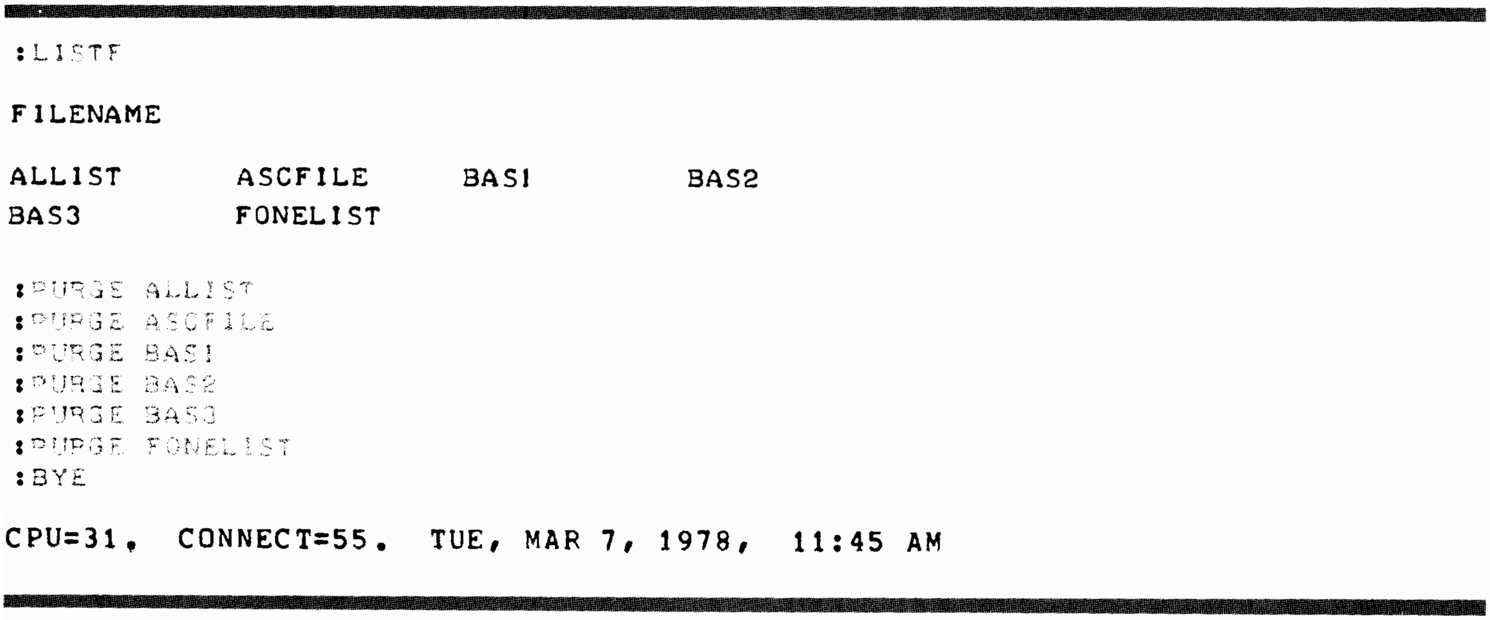


```
110      PRINT USING 120;B$[6],B$[1,4],C$
120      IMAGE 3X12A,"(EMPL#",X4A,"):",2X8A
130    NEXT N
140    STOP
150 DOEND
160 FOR N=1 TO 10
170    LINPUT #2,N;B$
180    IF A$=B$[8;LEN(A$)] THEN DO
190      RESTORE #1
200      ADVANCE #1;N-1,X
210      READ #1;C$
220      PRINT "PHONE NUMBER IS: ";C$
230      GOTO 30
240    DOEND
250 NEXT N
260 INPUT "  ***NO SUCH NAME***      NAME? ",A$
270 GOTO 40


NAME? ALL
THE NUMBERS ARE:
   E ALBERT     (EMPL# 0843):  930-1483
   J BELL       (EMPL# 0576):  861-7445
   P CARR       (EMPL# 0092):  955-2585
   Q DEXTER     (EMPL# 1438):  173-3790
   R EVANS      (EMPL# 1755):  543-8448
   G FAY        (EMPL# 0948):  285-9501
   S GOLD       (EMPL# 0663):  372-2503
   B HILL       (EMPL# 1349):  339-9721
   O HOWE       (EMPL# 0252):  561-5309
   A IDE        (EMPL# 1124):  356-5485
```

It is valuable to remember that the OUT= parameter may specify any ASCII file, including disc files. Thus, for example, the program listing or program output can be made accessible to the Editor, Sort, and language subsystems.

So don't stop here. The possibilities are unlimited!

Before logging off, now, purge all unneeded files. If necessary, use LISTF to see which files actually exist in your file group.



```
:LISTF

FILENAME

ALLIST          ASCFILE         BAS1            BAS2
BAS3            FONELIST

:PURGE ALLIST
:PURGE ASCFILE
:PURGE BAS1
:PURGE BAS2
:PURGE BAS3
:PURGE FONELIST
:BYE

CPU=31,  CONNECT=55.  TUE, MAR 7, 1978,  11:45 AM
```

summary

>SYSTEM :FILE PRINTER;DEV=LP :RESUME	makes line printer available for listings and output
>LIST,OUT=PRINTER	lists work area on line printer
>RUN, OUT=PRINTER	prints program output on line printer

section 7
using RPG/3000

summary of commands introduced in this section

:BUILD OUTPUT;REC=-80,1,F,ASCII;DEV=DISC	creates a permanent disc file OUTPT which contains 80 character fixed ASCII records with a blocking factor of 1
:EDITOR	calls the EDIT/3000 subsystem
/TEXT OUTPT,UNN	texts OUTPT file into work file of Editor
/LIST ALL	displays contents of work file
:FILE DATA=$STDIN	designates terminal as input device
:FILE OUTPT=$STDLIST	designates terminal as output device
:PREP $OLDPASS,PFILE	prepares object code (in $OLDPASS) into file named PFILE
:RESET DATA	erases previous file command issued for DATA file
:RPG MYPROG,UFILE	compiles source file MYPROG into system temporary file $OLDPASS
:RPG MYPROG,UFILE	compiles source file MYPROG into session-temporary file UFILE
:RPGGO MYPROG	compiles, prepares and executes source file MYPROG
:RPGGO MYPROG,$NULL	compiles, prepares and executes source file MYPROG; suppresses source listing by writing it to $NULL rather than to terminal
:RPGPREP MYPROG	compiles and prepares source file MYPROG into $OLDPASS
:RPGPREP MYPROG,PFILE	compiles and prepares source file MYPROG into session-temporary file PFILE
:RUN PFILE	runs prepared code which is in PFILE
:SAVE $OLDPASS,PFILE	saves object or program file (in $OLDPASS) into a permanent disc file named PFILE
:SAVE PFILE	makes PFILE a permanent file

using RPG/3000

This section of the terminal user's guide introduces you to RPG/3000 programming at the terminal. Before reading this section, you should have worked through Sections 1 and 2 of this guide and understand the MPE/3000 colon(:) prompt character and the EDITOR slash (/) prompt character.

RPG/3000 is a symbolic programming language compatible with IBM SYSTEM/3 RPG II and similar to IBM DOS RPG II, but with several enhancements. If you are familiar with IBM RPG or some other version of RPG, you will have little difficulty in acquainting yourself with RPG/3000. This portion of the terminal user's guide will show you how to enter RPG/3000 source statements on the terminal using EDIT/3000, how to compile your source RPG/3000 program using the RPG/3000 Compiler, and how to execute your program under the control of the MPE/3000 operating system.

entering RPG/3000 programs at the terminal

EDIT/3000 does not provide a special format setting for RPG/3000 as it does for COBOL/3000. However, there are several things you can do to simplify entering RPG programs at the terminal.

a. In default mode, EDIT/3000 allows you to use only 72 columns. The terminal display is 80 columns wide, but the Editor normally reserves 8 of these columns for the EDIT/3000 line numbers. Since RPG/3000 programming may require you to use 80 columns per source record (just as you may use 80 columns on a card), you need to increase the format from 72 to 80 columns. You do this by using the EDIT/3000 commands

```
/SET LENGTH=80
```

and

```
/SET RIGHT=80
```

These commands allow you to use 80 columns and to set the right margin at column 80. You should enter these commands immediately after calling EDIT/3000. Notice that columns 71 through 80, when you use them, appear on the next line. The default conditions of length=72 and right margin=72 resume as soon as you exit from the EDITOR.

b. Since RPG is a column-oriented language, you may want to devise some means of indicating the position of columns on the terminal display. For example, it is possible to enter a procedure which will draw the following template at the top of your display and lock it into place so that your RPG/3000 source statements will scroll up to and behind this template.

The template will help you locate columns, but will not affect your source statements. Entering the template-drawing procedure is not difficult and is described on the following pages.

```
7        8         1         2         3         4         5         6         7
123456789/1234567890123456789012345678901234567890123456789012345678901234567890
/
```

entering a template-drawing procedure

This section shows how to enter a template-drawing procedure which simplifies entering RPG/3000 source statements. THIS PROCEDURE FUNCTIONS ONLY ON HEWLETT-PACKARD TERMINALS IN THE 2600 FAMILY (for example, 2640A, 2641A, and so on). The procedure described clears the terminal display, draws a template at the top of the display, showing the position of the columns, and then locks the template into place, enabling you to scroll source statements up to the template. The template does not affect the entering of your source statements at all and allows you to scroll your statements up behind the template as you enter your RPG/3000 program.

The template-drawing procedure consists of five statements. Each statement begins with an upper case Q followed by quote marks (") and ends with a second pair of quote marks("). Some of the characters in the procedure instruct the terminal itself to perform specific actions, such as carriage return and line feed, and are instructions not normally used by the programmer.

EDITING AND LISTING THE TEMPLATE-DRAWING PROCEDURE

Any listing of the template-drawing procedure source statements must be done with the "DISPLAY FUNCTIONS" capability on (indicator light is on). Otherwise, the terminal tries to execute the functions in your procedure as it lists it. Try listing this procedure when DISPLAY FUNCTIONS is off and you will see what is meant. Turning DISPLAY FUNCTIONS on suppresses the execution of the terminal display control characters (escape codes). Similarly, any editing of the source statements should be done with the DISPLAY FUNCTIONS capability turned on.

ENTERING THE TEMPLATE-DRAWING PROCEDURE AT THE TERMINAL

1. Log on.
2. Call the Editor subsystem.
3. Enter the commands /SET LENGTH=92 and /SET RIGHT=92.
4. Select the ADD mode.
5. Latch the CAPS LOCK key to the ON (depressed) position.
6. Press the DISPLAY FUNCTIONS key (indicator light is on).
7. Enter the first line of the procedure by entering the indicated character or by pressing the indicated key or control button in the following list. All procedure statements begin in column one of the edit field.
   a. Q (upper case)
   b. " (quote marks)
   c. Press the ESC key.

      (Do not be alarmed if a blank is produced when you press the ESC key. It simply means that your terminal, although acknowledging the ESC signal, lacks the ability to draw the corresponding character on the display.)
   d. H
   e. Press the ESC key.
   f. J
   g. "
   h. Press the DISPLAY FUNCTIONS key (indicator light is off).
   I. Press RETURN.
8. Enter the second line of the procedure.
   a. Q
   b. "
   c. 7
   d. Press the Space Bar 8 times.
   e. 8
   f. Press the Space Bar 9 times.
   g. 1
   h. Repeat steps f and g six times, incrementing the digit entered in step g by one each time until you enter a 7.

i. "

j. Press RETURN.

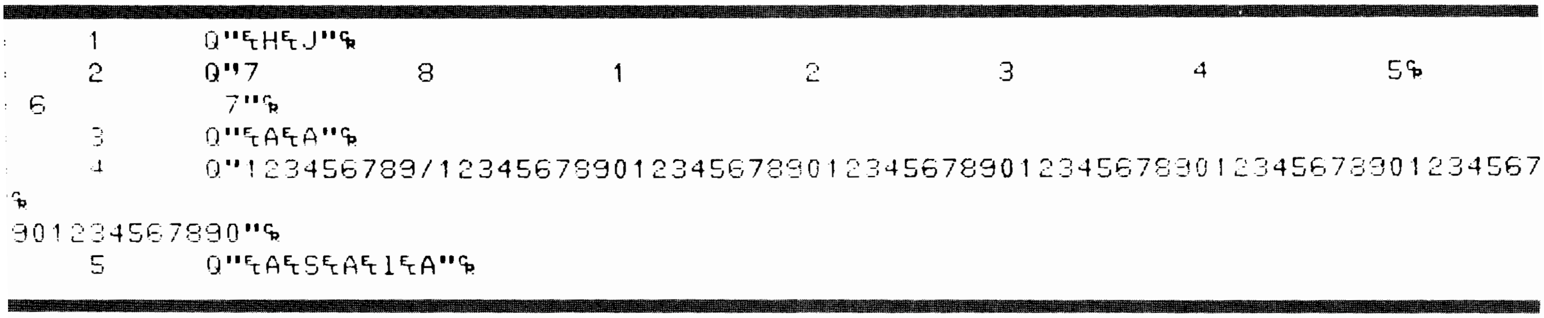
9. Enter the third line of the procedure.
   a. Press the DISPLAY FUNCTIONS key (indicator light is on).
   b. Q
   c. "
   d. Press the ESC key.
   e. A
   f. Press the ESC key.
   g. A
   h. "
   i. Press the DISPLAY FUNCTIONS key (indicator light is off).
   j. Press RETURN.

10. Enter the fourth line of the procedure.
    a. Q
    b. "
    c. 123456789/
    d. 1234567890 (Repeat 6 more times.)
    e. "
    f. Press RETURN.

11. Enter the fifth and last line of the procedure.
    a. Press the DISPLAY FUNCTIONS key (indicator light is on).
    b. Q
    c. "
    d. Press the ESC key.
    e. A
    f. Press the ESC key.
    g. S
    h. Press the ESC key.
    i. A
    j. Press the ESC key.
    k. l (You will have to unlatch the CAPS LOCK key to get the lower case L.)
    l. Press the ESC key.
    m. A
    n. "
    o. Press the DISPLAY FUNCTIONS key (indicator light is off).
    p. Press RETURN.

12. Type control Y to end input.

The template-drawing procedure should look something like



```
     1       Q"␛H␛J"␍
     2       Q"7           8           1           2           3           4           5␍
 6            7"␍
     3       Q"␛A␛A"␍
     4       Q"123456789/123456789012345678901234567890123456789012345678901234567
␍
901234567890"␍
     5       Q"␛A␛S␛A␛l␛A"␍
```

Blanks appear where you hit the ESC key if your terminal lacks the ability to draw the ESC (Escape) character. Similarly, the Carriage Return and Line Feed characters may also appear as blanks.

Save the procedure with a /KEEP command. You can now use the template-drawing procedure any time you are using the Editor. If you saved the procedure with the command

```
/
```

call the procedure with the command `

```
/
```

whenever you want to put the template on your display.

When you call the template-drawing procedure, you will see the MEMORY LOCK indicator light come on, indicating that memory lock is in effect. If you turn the MEMORY LOCK off (indicator light is off), the template can be scrolled off the screen like anything else.

entering a simple RPG/3000 program

Let's suppose you want to enter a simple RPG program. Log onto the system and call EDIT/3000 by typing EDITOR in response to the colon prompt character. Next enter the /SET LENGTH=80 and /SET RIGHT=80 commands to adjust the Editor record length for RPG/3000 source statements. Call your template-drawing procedure or enter the command /ADD 1 to begin entering RPG/3000 source statements directly.

Suppose you want to enter the following program which accepts a list price from the terminal or some other input device, accepts a percentage discount to be given on the list price, computes the resulting sales price, adds the sales tax (assumed to be 6% in this example), and returns the result to your terminal. The name of this program is DCOUNT (DISCOUNT).

```
0010
0020
0050
0060
0070
0080
0090
0100
0110
0120
01201
01202
0130
0140        10
0150        10
0160        10
0170
0175
0180
0190
0200
0205
```

```
0210
0220
0230
0240
0250
0260
0270
0280
0290
0300
0310
0320
0330
0340
0350
0360
0370
0380
0390
0400
0410
```

If you use the template-drawing procedure described earlier, you will notice that the cursor remains fixed as you scroll source statements up to the template. Remedy this situation by pressing the key that moves the cursor up (the key with the upward pointing arrow on the extreme right-hand side of your CRT terminal).

If you have no editing to do on your source statements, save the program in a disc file with the command

```
/
```

This command saves the program in the disc file MYPROG.

executing a simple RPG/3000 program

To run the RPG/3000 program you have just stored in a disc file, you must first get out of the Editor subsystem. Do this by entering the command

/

MPE returns an end of subsystem message and then a colon, indicating that it is ready for a command. Enter

:

and

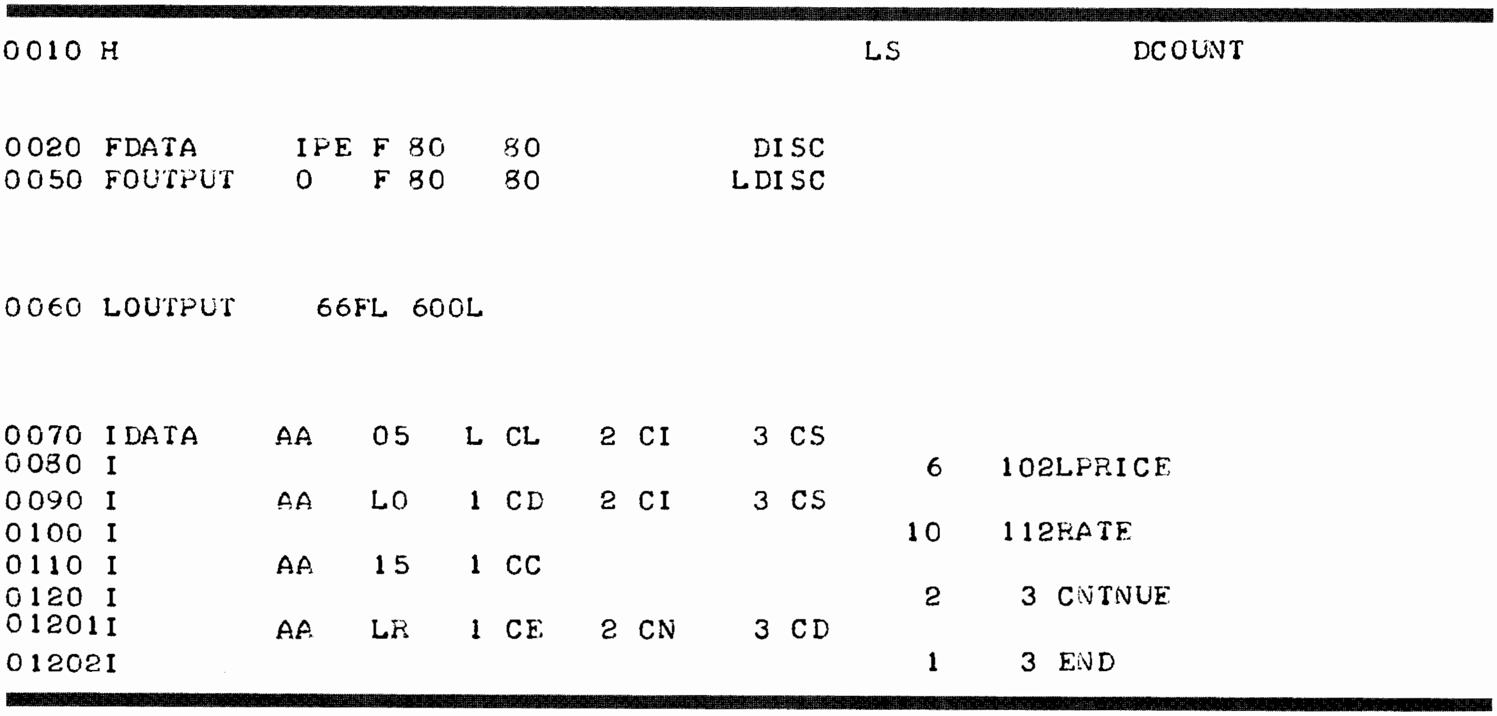
:

These MPE commands tell MPE that all input to the program will come from the standard input device and that all program output will go to the standard list device. Since you are working interactively at the terminal, these commands tell MPE that all program input will come from your terminal and that all output will go to your terminal. If you were working in a batch environment, data might come from a card reader or a disc drive and output might go to a line printer.

You are now ready to compile, prepare, and execute the sample RPG/3000 program. Type

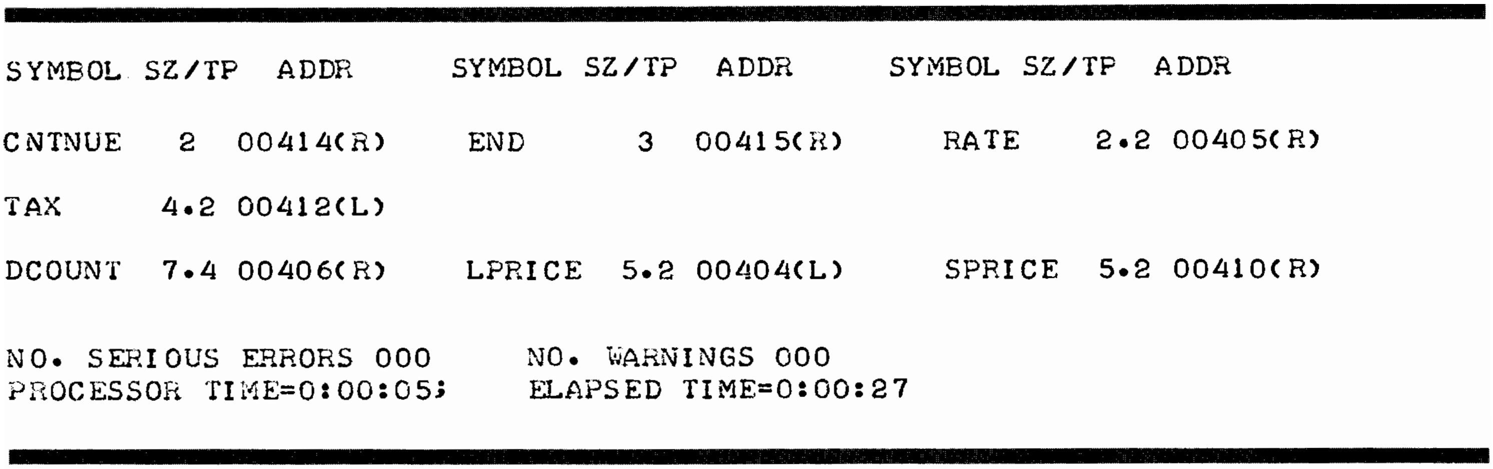
:

This command tells MPE to compile, prepare, and execute the source program in the disc file MYPROG. The RPG/3000 Compiler output appears as follows:



```
0010 H                                                   LS              DCOUNT


0020 FDATA      IPE F 80    80              DISC
0050 FOUTPUT    O   F 80    80             LDISC



0060 LOUTPUT     66FL 600L



0070 IDATA     AA    05    L CL   2 CI    3 CS
0080 I                                         6    102LPRICE
0090 I         AA    LO    1 CD   2 CI    3 CS
0100 I                                        10    112RATE
0110 I         AA    15    1 CC
0120 I                                         2      3 CNTNUE
01201I         AA    LR    1 CE   2 CN    3 CD
01202I                                         1      3 END
```

```
0130 C    10      LPRICE    MULT RATE      DCOUNT  74
0140 C    10      LPRICE    SUB  DCOUNT    SPRICE  52H
0150 C    10      SPRICE    MULT .06       TAX     42H
0160 C    10      SPRICE    ADD  TAX       SPRICE  52H

0170 OOUTPT    H 11      15
0175 O        OR         1P
0180 O                                  23 "ENTER LIST PRICE USING"
0190 O                                  39 "FOLLOWING FORMAT"
0200 O         D 11      15
0205 O        OR         1P
0210 O                                  19 "LIST=     (5 DIGIT "
0220 O                                  43 "NUMBER, RIGHT JUSTIFIED."
0230 O                                  59 " NO DECIMAL PT.)"
0240 O         D 11      05N10
0250 O                                  17 "ENTER % DISCOUNT "
0260 O                                  39 "USING FOLLOWING FORMAT"
0270 O         D 11      05N10
0280 O                                  22 "DISCOUNT=   (2 DIGITS)"
0290 O         D 11      10
0300 O                                  23 "SALES TAX OF 6% ASSUMED"
0310 O         H 11      10
0320 O                                  19 "DEBIT THIS AMT. TO "
0330 O                                  33 "CUSTOMER ACCT."
0340 O         D 11      10
0350 O                        SPRICE B  10 "  $0.  "
0360 O         D 11      10
0370 O                                  21 "IF YOU ARE DONE TYPE "
0380 O                                  26 "'END'"
0390 O         D 11      10
0400 O                                  22 "OTHERWISE TYPE 'C' TO "
0410 O                                  30 "CONTINUE"
```



```
SYMBOL SZ/TP  ADDR      SYMBOL SZ/TP  ADDR      SYMBOL SZ/TP  ADDR

CNTNUE   2  00414(R)    END      3  00415(R)     RATE    2.2 00405(R)

TAX     4.2 00412(L)

DCOUNT  7.4 00406(R)    LPRICE  5.2 00404(L)     SPRICE  5.2 00410(R)

NO. SERIOUS ERRORS 000      NO. WARNINGS 000
PROCESSOR TIME=0:00:05;     ELAPSED TIME=0:00:27
```

If serious errors in the program are discovered during compilation, the program terminates abnormally and the preparation and execution phases do not occur.

If compilation and preparation are successful and execution is initiated, DCOUNT asks you to enter a list price on your terminal.

Beginning in column 1, type

to indicate a list price of $100.00. Press Return.

DCOUNT will ask you for the discount. Type:

to indicate a discount of 20% off list price. Press Return. DCOUNT returns the total amount of the receipt with a sales tax of 6% figured in. Unless you wish to continue, enter

If you want to avoid the compilation and preparation phases the next time you run DCOUNT, you can save the program file of DCOUNT (the compiled and prepared version) by typing

:

$OLDPASS is the system default file where your program file is stored.

The second parameter, "PFILE", is a name supplied by you. Whenever you want to run DCOUNT in the future, you need only to log on and type

:

exercising compilation, preparation, and execution options

The MPE command RPGGO compiles, prepares, and executes your RPG/3000 source program. Compilation produces a USL (User Subprogram Library) file which contains relocatable binary modules (RBMs) which in turn contain the unlinked object code from the compilation of your source program. Your ULS file is passed to the MPE Segmenter which prepares your program by linking the relocatable binary modules from your USL file, producing a program file of re-entrant code segments. Your program file is passed to the MPE Loader which allocates a data stack, satisfies external references, and loads your program. Execution begins when MPE transfers control to your program.

MPE commands are structured so that you can exercise control over the compilation, preparation, and execution phases if you so desire. An option you will find useful is

```
:RPGGO MYPROG, $NULL
```

This command suppresses the listing of your program by sending the listing output to the $NULL file. The $NULL file is a special system file which allows unwanted output to be discarded.

You can perform an RPG/3000 compilation without preparation or execution by entering any of the following example commands. Each example is an illustration of the options available to you with the MPE command :RPG.

```
:RPG  MYPROG
```

(compilation only, USL file is system default file $OLDPASS listing produced on standard list device ($STDLIST))

```
:RPG MYPROG,UFILE
```

(compilation only USL file is user specified UFILE, listing produced on standard list device ($STDLIST))

```
:RPG  MYPROG,UFILE,$NULL
```

(compilation only, USL file is user specified UFILE, listing is discarded through system $NULL file)

```
:RPG  MYPROG,,$NULL
```

(compilation only, USL file is system default file $OLDPASS, listing is discarded through system $NULL file)

You can prepare the USL file you produce with the :RPG command with the MPE command :PREP. For example, if you compile a program with the command :RPG MYPROG, you can prepare the USL file and produce a program file named PFILE with the MPE command

```
:PREP $OLDPASS,PFILE
```

You must specify two files with the :PREP command – a USL file and the resulting program file. If your USL file is named UFILE, then you enter the command

```
:PREP UFILE,PFILE
```

A program file produced with the :PREP command is session-temporary unless you save it with the :SAVE command. (Otherwise the file disappears when you log off.) If you want to save the program file for use in future sessions, enter the MPE command

```
:SAVE PFILE
```

You must have previously assigned the name PFILE to the program file in the :PREP command.

You can specify compilation and preparation of your RPG/3000 program by using the MPE command :RPGPREP.

The program file produced by the command :RPGPREP is also session-temporary. If you want to save the program file for use in future sessions, enter

```
:SAVE $OLDPASS,PFILE
```

or

```
:SAVE PFILE
```

depending on whether or not you specified a name for the program file in the :RPGPREP command.

`:RPGPREP MYPROG`	(compilation and preparation, program file is system default file $OLDPASS, listing produced on standard list device ($STDLIST))
`:RPGPREP MYPROG,PFILE,$NULL`	(compilation and preparation, program file is user specified PFILE, listing is discarded through system file $NULL)

The following is a summary of MPE commands governing the compilation, preparation, and execution of RPG/3000 programs.

:RPGGO

1. Compiles, prepares, and executes.
2. USL file is not saved.
3. Program file may be saved (:SAVE).
4. Program may not reference Group or Public library routines, or Relocatable Library.

:RPGPREP

1. Compiles and prepares.
2. USL file is not saved.
3. Program file may be saved (:SAVE).
4. Program may reference library routines available to :RUN command at run time.

:RPG

1. Compiles only.
2. USL file may be saved (add a parameter).
3. USL file must be prepared into program file before execution can occur.

:PREPRUN

1. Prepares program file (from USL file) and executes same.
2. Program file may be saved (:SAVE).
3. Program may reference routines in Group, Public, or System Library, or Relocatable Library (LIB= or RL=).

:PREP

1. Prepares program file (from USL file) only.
2. Program file may be saved (:SAVE).
3. Program may reference Relocatable Library routines (RL=) plus, at run time, those libraries available to the :RUN command.

:RUN

1. Executes prepared program file only.
2. Program may reference routines in Group, Public, or System Library (LIB=).
3. Program may not reference Relocatable Library routines (RL=) unless resolved by :PREP.

reading from and writing to files in RPG/3000

Let's assume that you want to read input for DCOUNT from a disc file and write the output to a disc file. To do this, you have to build files for DCOUNT to read from and write to. Let's use the Editor to build the input file. Call the Editor and get into the ADD mode. Enter your input file data and then save it in a file named DATA.

```
:

/

     1
     2
     3
     4
     5
```

(Press Control Y to end input)

```
     6      •••

/
```

The UNN parameter on the /KEEP command tells the Editor to save the file in unnumbered format. You should do this because DCOUNT is not expecting to find any information in columns 73 through 80 where the Editor normally stores line numbers. If you include the UNN parameter on the /KEEP command, you must also include it when you reference the file with a /TEXT command. For example

```
/
```

At this point you must allocate a file to receive the output from DCOUNT. You allocate this file by using the BUILD command. The BUILD command is an MPE command and not an Editor command. You must therefore get out of the Editor subsystem before you can use the BUILD command. Do this and then enter

```
:BUILD OUTPT;REC=-80,1,F,ASCII;DEV=DISC
```

This command builds an ASCII disc file named OUTPT with 80 character records, a blocking factor of 1, and fixed length records. You now have both an input file and an output file.

Enter the commands

```
: [illegible]
         and
: [illegible]
```

These commands erase the FILE commands you entered to equate DATA to $STDIN and OUTPT to $STDLIST and ensures that your program reads from DATA and writes to OUTPT.

If you have saved a program file of DCOUNT, simply enter the command

```
:                              (name of program file)
```

If you saved only the DCOUNT source file, enter the command

```
:                                      (This command compiles, prepares, and runs DCOUNT without
                                       producing a listing.)
```

After your program is finished executing, you can examine the contents of OUTPUT. Call the Editor and then text in the file OUTPT into the Editor work area by entering the commands

```
:
/      [illegible]              (The UNN parameter tells the Editor
                                that OUTPUT does not have line
                                numbers.)
```

Now enter the command

```
/
```

The Editor lists the contents of OUTPT for you.

index

index

Part No. 03000-90121
Printed in U.S.A. 4/79
3MPEA.310.03000-90121

HEWLETT PACKAR