Order No. SH20-6162-0

Pascal/VS

Programmer's Guide

May 13th, 1980

Final Draft

IBM Internal Use Only

First Edition, May 1980
This is the first edition of SH20-6162, a new publication that applies to release 1.0 of the Pascal/VS Compiler (IUP program number 5796-PNQ).

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not available in your country. Such references or information must not be construed to mean that IBM intends to announce such products in your country.

Requests for copies of IBM publications should be made to the IBM branch office that serves you.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Pascal/VS Development, Department M48/D25, P. O. BOX 50020, San Jose, California 95150. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

PREFACE

This manual is a guide to the use of the Pascal/VS compiler. It explains how to compile and execute Pascal/VS programs, and describes the compiler and the operating system features which may be required by the Pascal/VS programmer. It does not describe the language implemented by the compiler.

RELATED PUBLICATIONS

- Pascal/VS Reference Manual, order number SH20-6163. This manual describes the Pascal/VS language.

- IBM Virtual Machine Facility/370: CMS Command and Macro Reference, order number GC20-1818. This manual describes the commands of the Conversational Monitor System (CMS) component of the IBM Virtual Machine Facility/370 with detailed reference information concerning command syntax and usage.

- IBM Virtual Machine Facility/370: CP Command Reference for General Users, order number GC20-1820. This manual describes the control processor commands of the IBM Virtual Machine Facility/370.

- OS/VS2 TSO Command Language Reference Manual, order number GC28-0646. This manual describes the commands of the Time Sharing Option of OS/VS2.

- OS/VS2 JCL, order number GC28-0692. This is a reference manual for the job control language of OS/VS2.

- OS/VS Linkage Editor and Loader, order number GC26-3813. This manual describes how to use the OS/VS2 linkage editor and loader.

- Time Sharing Option Display Support and Structured Programming Facility Version 2.2: Installation and Customization Guide, order number SH20-2402. This manual describes how to install and modify menus and command procedures of the Structured Programming Facility (SPF). Knowledge of the content of this manual is required to install the Pascal/VS SPF menus and procedures.

- OS/VS2 MVS Data Management Services Guide, order number GC26-3875. This manual describes the various data set access methods utilized by OS/VS2 and the OS simulation of CMS - VM/370.

LIST OF ILLUSTRATIONS

1.0 INTRODUCTION

The Pascal/VS compiler is a processing program which translates Pascal/VS source programs, diagnosing errors as it does so, into IBM System/370 machine instructions.

The compiler may be executed under the following operating system environments:

- OS/370 Batch (VS2 R3.7)
- Time Sharing Option (TSO) of OS/VS2
- Conversational Monitor System (CMS) of Virtual Machine Facility/370 (VM/370) Release 5 PLC 2.

This section applies only to those who are using Pascal/VS under the Conversational Monitor System (CMS) of Virtual Machine Facility/370 (VM/370). If you are not using CMS then you may skip this entire section.

For a description of the syntax notation used to describe commands, see "Command Syntax Notation" on page 117.

There are four steps to running a Pascal/VS program under CMS.

1. The program is compiled to produce an object module;
2. A load module is generated from the object module;
3. All files used within the program are defined using the FILEDEF command;
4. The load module is invoked.

```
PASCALVS | fn [maclibs...] ┌                    ┌PRINT  ┐                         ┐
         |                 │( [options...]      │NOPRINT│ [CONSOLE] [NOOBJ] [)]   │
         |                 └                    └DISK   ┘                         ┘
```

Figure 1. The PASCALVS command of CMS: invokes the Pascal/VS compiler.

2.1 HOW TO COMPILE A PROGRAM

2.1.1 Invoking the Compiler

The standard method of invoking the Pascal/VS compiler under CMS is by means of an EXEC called PASCALVS.

To compile a Pascal/VS program, the EXEC may be invoked in its simplest form by the command

```
PASCALVS fn
```

where "fn" is the file name of the program. The file type is always assumed to be "PASCAL".

The compiler translates a source program into object code, which it stores in a file. The name of this file is identical to the name of the source program. Its file type is "TEXT".

For example, to compile a program which resides in a file called "SORT PASCAL", the command would be:

```
PASCALVS SORT
```

If the compilation completes without errors, then the file named "SORT TEXT" will contain the resulting object code.

2.1.2 The PASCALVS Command

The generalized form of the PASCALVS command is illustrated in Figure 1. The operands of the command are defined as follows:

fn
is the file name of the source program; the assumed file type is "PASCAL".

maclibs...
are optional macro libraries required by the %INCLUDE facility. Up to eight may be specified.

options...
are compiler options, see "Compiler Options" on page 29.

The command options **PRINT**, **NOPRINT**, and **DISK** specify where the compiler listing is to be placed.

PRINT
specifies that the listing is to be spooled to the virtual printer. This is the default.

NOPRINT
specifies that the listing is to be suppressed. This option automatically forces the following three compiler options to become active:

- NOSOURCE
- NOXREF
- NOLIST

DISK
specifies that the listing is to be stored as a file on your A disk. The file is named "**fn LISTING**", where "fn" is the file name of the source program.

CONSOLE
This command option specifies that the console messages produced by

the compiler are be stored as a file on your A disk. The name assigned to the file is "fn CONSOLE". If **CONSOLE** is not specified, then the messages will be displayed on the terminal console.

NOOBJ
This command option suppresses the production of an object module by disabling the code generation phase of the compiler. This option is useful when you are using the compiler only as an error diagnoser.

For an explanation of the possible error messages and return codes produced from the EXEC, see "Messages from PASCALVS exec" on page 113.

2.1.3 The %INCLUDE Maclibs

The macro libraries (maclibs) that may be specified when invoking the PASCALVS command are those required by the **%INCLUDE** facility. When the compiler encounters an **%INCLUDE** statement within your program it will search the maclibs (in the order in which they were specified in the PASCALVS command) for the member named. When found, the maclib member becomes the input stream for the compiler. After the compiler has read the entire member, it will continue reading in the previous input stream (immediately following the **%INCLUDE** statement).

The default maclib named PASCALVS need not be specified. It is always implicitly provided as the last maclib in the search order.

2.1.4 Passing Compiler Options

Compile time options (see "Compiler Options" on page 29) are parameters that are passed to the compiler which specify whether or not a particular feature is to be active. A list of compiler options may be specified in the PASCALVS parameter list. The options list must be preceded by a left parenthesis "(".

For instance, to compile the program "TEST PASCAL" with the debug feature enabled and without a cross reference table, you would invoke the following command:

```
PASCALVS TEST ( DEBUG NOXREF
```

2.1.5 The Compiler Listing

The compiler generates a listing of the source program with such information as the lexical nesting structure of the program and cross reference tables. For a detailed description of the information on the source listing see "Source Listings" on page 33.

2.1.6 Compiler Diagnostics

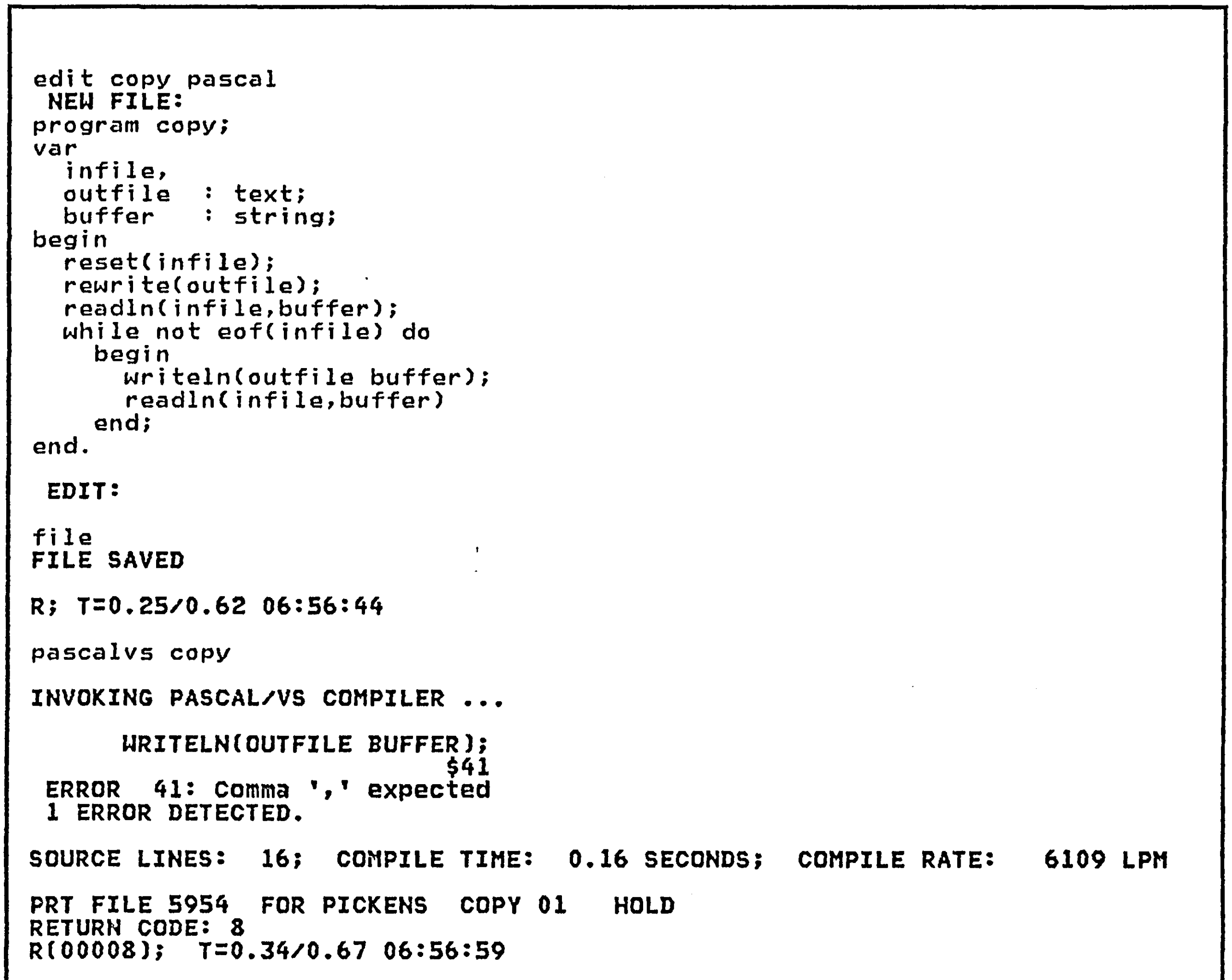
Any compiler-detected errors in your program will be displayed on your terminal console (or written to a disk file if the CONSOLE options is specified). The errors will also be indicated on your source listing at the lines where the errors were detected. The diagnostics are summarized at the end of the listing.

When an error is detected, the source line that was being scanned by the compiler is displayed on your console. Immediately underneath the printed line a dollar symbol ('$') is placed at each location where an error was detected. This symbol serves as a pointer to the approximate location where the error occurred within the source record.

Accompanying each error indicator is an error number. Beginning with the following line of your console a diagnostic message is produced for each error number.

For a synopsis of the compiler-generated messages see "Pascal/VS Compiler Messages" on page 95.

2.1.7 Sample Compilation



```
edit copy pascal
 NEW FILE:
program copy;
var
  infile,
  outfile   : text;
  buffer    : string;
begin
  reset(infile);
  rewrite(outfile);
  readln(infile,buffer);
  while not eof(infile) do
    begin
      writeln(outfile buffer);
      readln(infile,buffer)
    end;
end.

 EDIT:

file
FILE SAVED

R; T=0.25/0.62 06:56:44

pascalvs copy

INVOKING PASCAL/VS COMPILER ...

      WRITELN(OUTFILE BUFFER);
                          $41
 ERROR  41: Comma ',' expected
 1 ERROR DETECTED.

SOURCE LINES:  16;  COMPILE TIME:  0.16 SECONDS;  COMPILE RATE:   6109 LPM

PRT FILE 5954  FOR PICKENS  COPY 01   HOLD
RETURN CODE: 8
R(00008);  T=0.34/0.67 06:56:59
```

Figure 2. Sample compilation under CMS

```
PASCMOD    main [names ... ] [ ( options... [)]]
```

Figure 3. The PASCMOD command: generates a Pascal/VS load module.

2.2 HOW TO BUILD A LOAD MODULE

The PASCMOD EXEC generates load modules from Pascal/VS object code. If your program consists of just one source module (that is, you have no segment modules), a load module can be generated by simply invoking PASCMOD with the name of the program. For example, if a program named **SORT** was successfully compiled (which implies that "SORT TEXT" exists), then a load module may be generated with:

```
  PASCMOD SORT
```

The resulting module would be called "SORT MODULE". A load map is stored in "SORT MAP".

The general form of the PASCMOD command is shown in Figure 3.

The operands of the command are defined as follows:

main
is the name of the main program module.

names...
are the names of segment modules and text libraries (TXTLIB's) which are to be included. If a name "n" is specified and there are two files named n TEXT and n TXTLIB, then the TEXT file will be included _and_ the TXTLIB will be searched.

options...
is a list of options. (see "Module Generation Options.")

The resulting load module will be given the name "main MODULE A". The load map of the module will be stored in "main MAP A".

The Pascal/VS run time library resides in "PASCALVS TXTLIB"; PASCMOD implicitly appends this library to the list that you specify.

As an example, let us build a load module for a pre-compiled program which resides in three source modules: **MAIN**, **ASEG**, and **BSEG**. This program calls routines that reside in a txtlib called **UTILITY**. The following command would generate a load module called **MAIN MODULE**:

```
  PASCMOD MAIN ASEG BSEG UTILITY
```

2.2.1 Module Generation Options

The following are recognized as options to the PASCMOD command.

DEBUG
This option links the debugging routines into the load module so that the interactive debugger can be used. (See "Debug - Pascal/VS Interactive Debugger" on page 53.)

NAME name
This option specifies an alternate name for the load module. The resulting load module and map will have the name "name MODULE A" and "name MAP A".

2.2.2 Run time Libraries

Routines which make up the Pascal/VS runtime environment reside in a text library called "PASCALVS TXTLIB". It must be present in order to resolve the linkages from the program being prepared for execution.

The name of the txtlib which contains the runtime Debug support is "PASDEBUG TXTLIB". (see "Debug - Pascal/VS Interactive Debugger" on page 53 for a description of Debug).

```
FILEDEF SYSIN DISK INPUT DATA
FILEDEF SYSPRINT PRINTER (LRECL 133 RECFM VA
FILEDEF OUTPUTFI DISK OUTPUT DATA (RECFM F LRECL 4
FILEDEF OUTPUT TERMINAL (RECFM F LRECL 80
FILEDEF INPUT TERMINAL (RECFM V LRECL 80
```

Figure 4. Examples of CMS file definition commands

2.3 HOW TO DEFINE FILES

Before you invoke the generated load module, you must first define the files that your program requires. This is done with the **FILEDEF** command.

The first parameter of the FILEDEF command is the file's ddname. The ddname to be associated with a particular file variable in your program is normally the name of the file variable itself, truncated to eight characters.

For example, the ddnames for the variables declared within the Pascal declaration below would be SYSIN, SYSPRINT, and OUTPUTFI, respectively.

```
var
   SYSIN,
   SYSPRINT    : TEXT;
   OUTPUTFILE  : file of
                    INTEGER;
```

The text file named OUTPUT receives the execution time error diagnostics. You must always define this file prior to executing any Pascal/VS program. This file is often assigned to the terminal.

The text file named INPUT is required by the interactive debugger ("Debug - Pascal/VS Interactive Debugger" on page 53) to be assigned to the terminal.

If a particular file is to be opened for input, attributes such as LRECL, BLKSIZE, and RECFM are obtained from the (presumably) already existing file.

For the case of files to be opened for output, the LRECL, BLKSIZE, or RECFM will be assigned default values if not specified. For a description of the defaults see "Data Set DCB Attributes" on page 39.

The FILEDEF commands required for each of the three file variables in the example above and for INPUT and OUTPUT could be as shown in Figure 4.

2.4 HOW TO INVOKE THE LOAD MODULE

After the module has been created and the files defined, you are ready to execute the program. This is done by invoking the module.

If your program expects to read a parameter list via the PARMS function, the list must follow the module name:

```
modname [parms...]
```

where "**modname**" is the name of the load module and "**parms**" are the parameters (if any) being passed.

Run time options are also passed as a parameter list. To distinguish runtime parameters being passed to the Pascal/VS environment from those that your program will read (via the PARMS function), the runtime parameter list must be terminated with a slash "/". The program parameters, if any, must follow the "/".

```
modname [ [rtparms.../] [parms...] ]
```

2.4.1 Run Time Options

The following options enable features in the Pascal/VS run time environment in which your program will be executing.

COUNT
This option causes instruction frequency information to be collected during program execution. This option will only have an effect if the program was both compiled and loaded with the DEBUG option.

DEBUG
The DEBUG option causes the interactive debugger, Debug ("Debug - Pascal/VS Interactive Debugger" on page 53) to gain initial control when you invoke your program. Note: this option is valid only if the load module was generated with the DEBUG option ("Module Generation Options" on page 6).

This section describes how to compile a Pascal/VS program under the Time Sharing Option (TSO) of OS/VS2. If you are not using TSO to run the compiler, you may skip this section.

Refer to "Command Syntax Notation" on page 117 for a description of the syntax notation used to describe commands.

There are four steps to running a Pascal/VS program.

1. The program is compiled to form an object module;
2. A load module is generated from the object module;
3. All data sets used within the program are allocated;
4. The load module is invoked.

CLIST NAME	OPERANDS
PASCALVS	data-set-name [compiler-options-list] [OBJECT(dsname) NOOBJECT] [PRINT(*) PRINT(dsname) SYSPRINT(sysout-class) NOPRINT] [CONSOLE(*) CONSOLE(dsname)] [LIB(dsname-list) NOLIB]

Figure 5. PASCALVS CLIST syntax.

3.1 HOW TO COMPILE A PROGRAM

3.1.1 Invoking the Compiler

The Pascal/VS compiler is invoked under TSO by means of a CLIST. A sample CLIST named PASCALVS is provided to compile a Pascal/VS program.

data-set-name
specifies the name of the primary input data set in which contains the source program to be compiled. This can be either a fully qualified name (enclosed in single quotation marks) or a simple name (to which the user identification will be prefixed and the qualifier "PASCAL" will be suffixed). This must be the first operand specified.

compiler-options-list
specifies one or more compiler options. See "Compiler Options" on page 29.

OBJECT(dsname)
specifies that the object module produced by the compiler is to be written to the data set named in the parentheses. This can be either a fully qualified name (enclosed within triple quotation marks '''...''')[1] or a simple name (to which the identification qual-

[1] Triple quotes are required because the CLIST processor removes the outer quotes within a keyword sub-operand list.

ifier will be prefixed and the qualifier "OBJ" suffixed).

NOOBJECT
specifies that no object module is to be produced. The compiler will diagnose errors only.

If neither OBJ nor NOOBJ is specified then object module produced by the compiler will be written to a default data set. If the data set specified in the first operand contains a descriptive qualifier of "PASCAL", the CLIST will form a data set name for the object module by replacing the descriptor qualifier of the input data set with "OBJ". If the descriptive qualifier is not "PASCAL", then you will be prompted for the object module data set name.

If the first operand of PASCALVS specifies the member of a partitioned data set, the member name will be ignored - the generated data set name will be based on the name of the partitioned data set.

As an example, given that the user identification is ABC, the following commands will produce object modules with the name shown.

PASCALVS SORT
object module: 'ABC.SORT.OBJ'

PASCALVS 'DEF.PDS.PASCAL(MAIN)'
object module: 'DEF.PDS.OBJ'

PASCALVS 'ABC.PROG.PAS'
user prompted for object
module name

PRINT(*)
specifies that the compiler listing is to be written at the terminal; no other copy will be available.

PRINT(dsname)
specifies that the compiler listing is to be written on the data set named in the parentheses. This can be either a fully qualified name (enclosed within triple quotation marks '''...''')[2] or a simple name (to which the identification qualifier will be prefixed and the qualifier "LIST" suffixed).

SYSPRINT(sysout-class)
specifies that the compiler listing is to be written to the sysout class named in parentheses.

NOPRINT
specifies that the compiler listing is not to be produced. This operand activates the following compiler options:
NOSOURCE, NOXREF, NOLIST

CONSOLE(*)
specifies that the compiler generated messages are to be displayed on the terminal console. This is the default.

CONSOLE(dsname)
specifies that the compiler generated messages are to be written to the data set named in the parentheses. This can be either a fully qualified name (enclosed within triple quotation marks '''...''') or a simple name (to which the identification qualifier will be prefixed and the qualifier "CONSOLE" suffixed).

LIB(dsname-list)
specifies that the %INCLUDE facility is being utilized. Within the parentheses is a list of the names of one or more partitioned data sets that are to be searched for members to be included within the input stream.

If the list contains more than one name, the entire list must be enclosed within quotes. Any fully qualified name within the quoted list must be enclosed in double quotes ''...''.

See "Using the %INCLUDE Facility" on page 11.

NOLIB
specifies that no %INCLUDE libraries are required. This is the default.

<u>Example 1</u>

Operation: Invoke the Pascal/VS compiler to process a Pascal/VS program

Known: User-identification is ABC

Data set containing the program is named ABC.SORT.PASCAL

The compiler listing is to be directed to the printer.

Default options and data set names are to be used.

```
PASCALVS SORT SYSPRINT(A)
```

[2] Triple quotes are required because the CLIST processor removes the outer quotes within a keyword sub-operand list.

Example 2

Operation: Invoke the Pascal/VS compiler to process a Pascal/VS program

Known: User-identification is XYZ

Data set containing the program is named ABC.TEST.PASCAL

The compiler listing is to be directed to a data set named XYZ.TESTLIST.LIST.

The long version of the cross reference listing is preferred.

Default options and data set names are to be used for the rest.

```
PASCALVS 'ABC.TEST.PASCAL' +
     XREF(LONG),PRINT(TESTLIST)
```

3.1.2 Using the %INCLUDE Facility

If the %INCLUDE facility is used within the source program, then the names of the library or libraries to be searched must be listed within the LIB parameter of the PASCALVS CLIST.

The standard include library supplied by IBM is called[3]

"SYS1.PASCALVS.MACLIB"

This library must be specified in the LIB list if your program contains an %INCLUDE statement for one of the IBM supplied members.

When the compiler encounters an %INCLUDE statement within the source program, it will search the partitioned data set(s) in the order specified for the member named within the statement. When found, the member becomes the input stream for the compiler. After the compiler has read the entire member, it will continue reading from the previous input stream immediately following the %INCLUDE statement.

Example 1

Operation: Invoke the Pascal/VS compiler to process a Pascal/VS program which utilizes the %INCLUDE facility.

Known: User-identification is P123

Data set containing the program is named

'P123.MAIN.PASCAL'

The source to be included is stored in two partitioned data sets by the names of

'P123.PASLIB'
'SYS1.PASCALVS.MACLIB'.

Default options and data set names are to be used for the rest.

```
PASCALVS MAIN LIB('PASLIB,+
    ''SYS1.PASCALVS.MACLIB''')
```

3.1.3 Compiler Diagnostics

By default, compiler diagnostics are displayed on your terminal. If the **CONSOLE(dsname)** operand appears on the PASCALVS command, then the diagnostics will be stored in a data set. The errors will also be indicated on your source listing at the lines where the errors were detected. The diagnostics are summarized at the end of the listing.

When an error is detected, the source line that was being scanned by the compiler is printed on your terminal (or to the **CONSOLE** data set). Immediately underneath the printed line, a dollar symbol ('$') is placed at each location where an error was detected. This symbol serves as a pointer to indicate the approximate location where the error occurred within the source record.

Accompanying each error indicator is an error number. Beginning with the following line of your console a diagnostic message is produced for each error number.

For a synopsis of the compiler generated messages see "Pascal/VS Compiler Messages" on page 95.

[3] The high-level qualifier name (SYS1) may be different at your installation.

3.2 HOW TO BUILD A LOAD MODULE

To generate a load module from a Pascal/VS object module, you may use either the TSO LINK command or a CLIST named "PASCMOD" (Figure 6 on page 13). The CLIST performs the same function as the LINK command except that it will automatically include the Pascal/VS runtime library in generating the load module. Also, if the debugger is to be utilized, the CLIST will include the Pascal/VS debug library. (A complete description of the LINK command is contained in the TSO Command Language Reference Manual.)

Every Pascal/VS object module contains references to the runtime support routines. These routines are stored in a library called[4]

"SYS1.PASCALVS.LOAD"

This library must be linked into a Pascal/VS object module in order to resolve all external references properly. If the PASCMOD CLIST is used, this library is included automatically.

If the interactive debugger is to be utilized, then the library containing the debug environment must be included in the linking. The name of this library is[4]

"SYS1.PASDEBUG.LOAD"

This library must appear ahead of the runtime library in search order. If the PASCMOD CLIST is used, this library will be included if the option DEBUG is specified.

If more than one object module is being linked together, then an entry point should be specified by means of a linkage editor control card. The name of the entry point for any Pascal/VS program is AMPXSTRT.

[4] The high-level qualifier name (SYS1) may be different at your installation.

CLIST NAME	OPERANDS
PASCMOD	data-set-name or * [OBJECT('dsname-list')] [DEBUG] [LOAD(dsname)] [PRINT(*) PRINT(dsname) NOPRINT] [LET NOLET] [XCAL NOXCAL] [LIB('dsname-list')] [FORTLIB] [COBLIB] [MAP NOMAP] [NCAL NONCAL] [LIST NOLIST] [XREF NOXREF] [REUS NOREUS] [REFR NOREFR] [SCTR NOSCTR] [OVLY NOOVLY] [RENT NORENT] [NE NONE] [OL NOOL] [DC NODC] [TEST NOTEST] [NOTERM TERM] [SIZE('integer1 integer2')] [DCBS(blocksize)] [AC(authorization-code)]

Figure 6. The TSO PASCMOD CLIST description.

data-set-name
specifies the name of a data set containing a Pascal/VS object module and/or linkage editor control cards. If more than one object module is to be linked, then their names should appear in the **OBJECT** sub-parameter list.

You may substitute an asterisk (*) for the data set name to indicate that you will enter control statements from your terminal. The system will prompt you to enter the control statements. A null line indicates the end of your control statements.

OBJECT('dsname-list')
specifies a list of data sets which contain object modules to be included in the link edit. Because of CLIST restrictions, the list must be enclosed in single quotes; fully qualified names within the list must be enclosed in double quotes (''...'').

LIB('dsname-list')
specifies one or more names of library data sets to be searched by the linkage editor to locate load modules referred to by the module being processed, that is, to resolve external references. The name of the Pascal/VS runtime library is implicitly appended to the end of this list; you need not specify it.

Because of CLIST restrictions, the list must be enclosed in single quotes; fully qualified names within the list must be enclosed in double quotes (''...'').

DEBUG
specifies that the Pascal/VS interactive debugger is to be utilized on the resultant load module. This will cause the Pascal/VS debug library to be included among the libraries to be searched to resolve external references.

All other operands of the PASCMOD CLIST are identical to their counterparts in the LINK command as described in the TSO Command Language Reference Manual.

Example

Operation: Create a load module from a compiled Pascal/VS program consisting of three object modules.

Known: User-identification is ABC.
Data sets containing the three object modules:

```
ABC.SORT.OBJ
ABC.SEG1.OBJ
ABC.SEG2.OBJ
```

The resulting load module is to be stored as a member named SORT in a data set named ABC.PROGS.LOAD

(The user's input is in lower case; the system replies are **high-lighted.**)

```
pascmod * load(progs(sort)) +
  object('sort,seg1,seg2')
ENTER CONTROL CARDS
 entry ampxstrt

READY
```

```
ATTR F80 LRECL(80) BLKSIZE(80) RECFM(F)
ALLOC DDNAME(SYSIN) DSNAME(INPUT.DATA) SHR
ALLOC DDNAME(SYSPRINT) SYSOUT(A)
ALLOC DDNAME(OUTPUTFI) DSNAME(OUTPUT.DATA) NEW SPACE(100) BLOCK(3120)
ALLOC DDNAME(OUTPUT) DSNAME(*) USING(F80)
ALLOC DDNAME(INPUT) DSNAME(*) USING(F80)
```

Figure 7. Examples of TSO data set allocation commands

3.3 HOW TO DEFINE FILES

Before you invoke the generated load module, you must first define the files that your program requires. This is done with the ALLOC command.

The ddname to be associated with a particular file variable in your program is normally the name of the variable itself, truncated to eight characters.

For example, the ddnames for the variables declared within the Pascal declaration below would be SYSIN, SYSPRINT, and OUTPUTFI, respectively.

```
var
  SYSIN,
  SYSPRINT    : TEXT;
  OUTPUTFILE  : file of
                  INTEGER;
```

The text file named OUTPUT receives the execution time error diagnostics. You must always allocate the ddname OUTPUT prior to executing any Pascal/VS program. This ddname is often assigned to the terminal.

The text file named INPUT is required by the interactive debugger (see "Debug - Pascal/VS Interactive Debugger" on page 53) to be assigned to the terminal.

For the case of files to be opened for output, the LRECL, BLKSIZE, or RECFM will be assigned default values if not specified via the ATTR command. For a description of the defaults see "Data Set DCB Attributes" on page 39.

The ALLOC commands required for each of the three file variables in the example above and for INPUT and OUTPUT could be as shown in Figure 7.

CALL	dsname[(member)] ['[options/] [parms]']

Figure 8. The TSO CALL command to invoke a load module

3.4 INVOKING THE LOAD MODULE

After the module has been created and the files defined, you are ready to execute the program. This is done by the CALL command (see Figure 8). The operands of the CALL command are as follows.

dsname(member)
specifies the name of a partitioned data set and the member where the load module to be invoked is stored. If the member name is omitted, then the member "TEMPNAME" will be the load module invoked.

dsname may be either a simple name (to which the user identification is prefixed and the qualifier "LOAD" is suffixed), or a fully qualified name in quotes.

options
specifies one or more run time options separated by either a comma or a blank. (See "Run Time Options.").

parms
specifies a parameter string which is to be passed to the program. The parameter string is retrieved from within the program by the PARMS function.

The total length of the quoted string (**options** plus **parms**) must not exceed 100 characters.

3.4.1 Run Time Options

The following options enable features in the Pascal/VS run time environment in which your program will be executing.

COUNT
This option causes instruction frequency information to be collected during program execution. This option will only have an effect if the program was compiled with the DEBUG option and linked with the Debug library[5].

DEBUG
The DEBUG option causes the interactive debugger to gain initial control when you invoke your program. For a description of the debugger see "Debug - Pascal/VS Interactive Debugger" on page 53.

Note: this option is valid only if the load module was linked with the Debug library[5].

[5] The Debug library will be included if the PASCMOD CLIST is invoked with DEBUG specified. See "How to Build a Load Module" on page 12.

3.5 SAMPLE TSO SESSION

```
READY

 pascalvs lander sysprint(a) list

INVOKING PASCAL/VS R1.0
NO COMPILER DETECTED ERRORS
SOURCE LINES:  47;  COMPILE TIME:  0.19 SECONDS;  COMPILE RATE:  15032

READY

 pascmod lander load(programs(lander))
READY

 alloc ddname(input) dsname(*)
READY

 alloc ddname(output) dsname(*)
READY

 call programs(lander) 'parms go here'
```

Figure 9. Sample TSO session of a compile, link-edit, and execution.

Figure 9 is an example of a TSO session which compiles an already existing source module, link edits it, and executes it. The commands entered from the terminal are in lower case; those produced by the system are in upper case and **high-lighted**.

This section describes how to compile and execute Pascal/VS programs in an OS Batch environment. If you are not using the compiler under OS Batch then you may skip this section.

4.1 JOB CONTROL LANGUAGE

Job control language (JCL) is the means by which you define your jobs and job steps to the operating system; it allows you to describe the work you want the operating system to do, and to specify the intput/output facilities you require.

The JCL statements which are essential to run a Pascal/VS job are as follows:

- JOB statement, which identifies the start of the job.
- EXEC statement, which identifies a job step and, in particular, specifies the program to be executed, either directly or by means of a cataloged procedure (described subsequently).
- DD (data definition) statement, which defines the input/output facilities required by the program executed in the job step.
- /* (delimiter) statement, which separates data in the input stream from the job control statements that follow this data.

A full description of job control language is given in the publication OS/VS2 JCL (GC28-0692).

4.2 HOW TO COMPILE AND EXECUTE A PROGRAM

The job control statements shown in Figure 10 on page 20 are sufficient to compile and execute a Pascal/VS program consisting of one module. This program uses only the standard files INPUT and OUTPUT. For a more generalized description of input/output refer to "How to Access Data Sets" on page 27 and "Using Input/Output Facilities" on page 39. Any options to be passed to the compiler are placed within the OPTIONS parameter of the EXEC statement.

```
//EXAMPLE JOB
//STEP1   EXEC PASCCG,OPTIONS=''
//PASC.SYSIN DD *
 program EXAMPLE(INPUT,OUTPUT);
 var
   A, B: REAL;
 begin
   RESET(INPUT);
   while not EOF do
     begin
       READLN(A,B);
       WRITELN(' SUM     = ',A+B);
       WRITELN(' PRODUCT = ',A*B);
     end
 end.
/*
//GO.INPUT DD *
 3.0 4.0
 3.14159 1.414
 1.0E-10 2.0E-10
 -10.0 102.0
/*
```

Figure 10. Sample JCL to run a Pascal/VS program

In the sample JCL, "EXAMPLE" is the name of the job. The job name identifies the job within the operating system; it is essential. The parameters required in the JOB statement depend on the conventions established for your installation.

The EXEC statement invokes the IBM supplied cataloged procedure named PASCCG. When the operating system encounters this name, it replaces the EXEC statement with a set of JCL statements that have been written previously and cataloged in a system library. The cataloged procedure contains three steps:

PASC The first pass of the compiler processes the Pascal/VS program and translates it into an intermediate form that will serve as input for the next step.

PASCT The second pass of the compiler reads in the intermediate code produced from the first pass and produces an object module.

GO The LOADER is invoked to process the object module by loading it into memory and including the appropriate runtime library routines. The resulting executable program is immediately executed.

The DD statement named "PASC.SYSIN" indicates that the program to be processed in procedure step PASC follows immediately in the card deck. "SYSIN" is the name that the compiler uses to refer to the data set or device on which it expects to find the program.

The delimiter statement /* indicates the end of the data.

The DD statement named "GO.INPUT" indicates that the data to be processed by the program (in procedure step GO) follows immediately in the card deck.

4.3 CATALOGED PROCEDURES

Regularly used sets of job control statements can be prepared once, given a name, stored in a system library, and the name entered in the catalog for that library. Such a set of statements is termed a cataloged procedure. A cataloged procedure comprises one or more job steps (though it is not a job, because it must not contain a JOB statement). It is included in a job by specifying its name in an EXEC statement instead of the name of a program.

Several IBM-supplied cataloged procedures are available for use with the Pascal/VS compiler. It is primarily by means of these procedures that a Pascal/VS job will be run.

The use of cataloged procedures saves time and reduces errors in coding frequently used sets of job control statements. If the statements in a cataloged procedure do not match your requirements exactly, you can easily modify them or add new statements for the duration of a job.

It is recommended that each installation review these procedures and modify them to obtain the most efficient use of the facilities available and to allow for installation conventions.

4.4 IBM SUPPLIED CATALOGED PROCEDURES

The standard cataloged procedures supplied for use with the Pascal/VS compiler are:

PASCC	Compile only
PASCCG	Compile, load-and-execute
PASCCL	Compile and link edit
PASCCLG	Compile, link edit, and execute

These cataloged procedures do not include a DD statement for the source program; you must always provide one. The DDname of the input data set is SYSIN; the procedure step name which reads the input data set is PASC. For example, the JCL statements that you might use to compile, link edit, and execute a Pascal/VS program is as follows:

```
  //JOBNAME JOB
  //STEP1   EXEC PASCCLG
  //PASC.SYSIN DD *
                    .
                    .
                    .
      (insert Pascal/VS program here
       to be compiled)
                    .
                    .
                    .
  /*
```

The listings and diagnostics produced by the compiler are directed to the device or data set associated with the DDname SYSPRINT. Each cataloged procedure routes DDname SYSPRINT to the output class where the system messages are produced (SYSOUT=*).

The object module produced from a compilation is normally placed in a temporary data set and erased at the end of the job. If you wish to save it in a cataloged data set or punch it to cards then the DDname SYSPUNCH in procedure step PASCT must be overridden. For example, to compile a program stored in data set

```
  "T123.SORT.PASCAL"
```

and to store the resulting object module in a data set named

```
  "T123.SORT.OBJ"
```

the following JCL might be employed:

```
//JOBNAME JOB
//STEP1   EXEC PASCC
//PASC.SYSIN DD DSN=T123.SORT.PASCAL,
//               DISP=SHR
//PASCT.SYSPUNCH DD DSN=T123.SORT.OBJ,
//               UNIT=TSOPACK,
//               DISP=(NEW,CATLG)
```

```
//PASCC   PROC SYSOUT=*,OPTIONS=,INCLLIB='SYS1.PASCALVS.MACLIB',
//             LINKLIB='SYS1.PASCALVS.LINKLIB'
//*
//*    P A S C
//*
//PASC     EXEC  PGM=PASCALL,PARM='&OPTIONS'
//STEPLIB   DD  DSN=&LINKLIB,DISP=SHR
//SYSPRINT DD  SYSOUT=&SYSOUT,DCB=(RECFM=VBA,LRECL=133,BLKSIZE=685)
//OUTPUT    DD  SYSOUT=&SYSOUT
//SYSTERM   DD  DUMMY
//SYSMSGS   DD  DSN=SYS1.PASCALVS.MESSAGES,DISP=SHR
//SYSLIB    DD  DSN=&INCLLIB,DISP=SHR
//          DD  DSN=SYS1.PASCALVS.MACLIB,DISP=SHR
//SYSBU     DD  UNIT=SYSDA,DISP=(NEW,PASS),
//              SPACE=(TRK,(2,5))
//SYSXREF   DD  UNIT=SYSDA,DISP=(NEW,DELETE),
//              SPACE=(TRK,(2,5))
//SYSPUNCH DD  SYSOUT=&SYSOUT
//SYSLIST   DD  UNIT=SYSDA,DISP=(NEW,PASS),
//              SPACE=(TRK,(2,5))
//*
//*    P A S C T
//*
//PASCT    EXEC  PGM=PASCALT,COND=(8,LE,PASC),PARM='&OPTIONS'
//STEPLIB   DD  DSN=&LINKLIB,DISP=SHR
//SYSPRINT DD  SYSOUT=&SYSOUT,DCB=(RECFM=VBA,LRECL=133,BLKSIZE=685)
//OUTPUT    DD  SYSOUT=&SYSOUT
//SYSTERM   DD  DUMMY
//INPUT     DD  DUMMY
//SYSIN     DD  DSN=*.PASC.SYSBU,DISP=(OLD,DELETE)
//SYSPUNCH DD  DSN=&&LOADSET,UNIT=SYSDA,DISP=(MOD,PASS),
//              SPACE=(TRK,(2,5)),
//              DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DSORG=PS)
//LOG       DD  SYSOUT=&SYSOUT
//SYSLIST   DD  DSN=*.PASC.SYSLIST,DISP=(MOD,DELETE)
//SYSUT1    DD  UNIT=SYSDA,DISP=(NEW,DELETE),
//              SPACE=(TRK,(2,5)),
//              DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DSORG=PS)
//SYSUT2    DD  UNIT=SYSDA,DISP=(NEW,DELETE),
//              SPACE=(TRK,(2,5)),
//              DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DSORG=PS)
```

Figure 11. Cataloged procedure PASCC

4.4.1 Compile Only (PASCC)

This cataloged procedure (Figure 11) compiles one Pascal/VS source module and produces an object module. It consists of two steps, PASC and PASCT, which are common to all of the cataloged procedures described in this chapter.

Step PASC reads in the source module, diagnoses errors, produces a listing, and translates the source into an intermediate form which it passes to the PASCT step. The PASCT step produces the object module and writes it to the data set associated with DDname SYSPUNCH.

The DD statement for the object module defines a temporary data set named &&LOADSET. The term MOD is specified in the DISP parameter and as a result, if the procedure PASCC is invoked several times in succession for different source modules, &&LOADSET will contain a concatenation of object modules. The linkage editor and loader will accept such a data set as input.

```
//PASCCG  PROC SYSOUT=*,OPTIONS=,INCLLIB='SYS1.PASCALVS.MACLIB',
//             LKLBDSN='SYS1.PASCALVS.LOAD',
//             LINKLIB='SYS1.PASCALVS.LINKLIB'
//PASC    EXEC PGM=PASCALL,PARM='&OPTIONS'

          ... (this step is identical to the PASC step in procedure PASCC)

//PASCT   EXEC PGM=PASCALT,PARM='&OPTIONS'

          ... (this step is identical to the PASCT step in procedure PASCC)

//GO      EXEC PGM=LOADER,COND=((8,LE,PASC),(8,LE,PASCT)),
//             PARM='EP=AMPXSTRT'
//SYSLIB    DD DSN=&LKLBDSN,DISP=SHR
//          DD DSN=SYS1.PASCALVS.LOAD,DISP=SHR
//SYSLIN    DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//SYSLOUT   DD SYSOUT=&SYSOUT
//SYSPRINT  DD SYSOUT=&SYSOUT
//OUTPUT    DD SYSOUT=&SYSOUT,DCB=(RECFM=VBA,LRECL=133,BLKSIZE=685)
//INPUT     DD DUMMY,DCB=(RECFM=V,LRECL=256,BLKSIZE=260)
```

Figure 12. Cataloged procedure PASCCG

4.4.2 Compile, Load, and Execute (PASCCG)

In this cataloged procedure (Figure 12), the first two steps compile a Pascal/VS source module to produce an object module. In the third step (named GO), the loader is executed; this program processes the object module produced by the compiler and executes the resultant executable program immediately.

The DD statement labeled SYSLIB in step GO describes the libraries from which external references are to be resolved. If you have a library of your own from which you would like external references to be resolved, then pass its name in the LKLBDSN operand.

Object modules from previous compilations may also be included in the loader's input stream by concatenating them in the SYSLIN DD statement.

As an example, a program in a data set named "DOE.SEARCH.PASCAL" needs to be compiled and then loaded with an object module named "DOE.SORT.OBJ". In addition, several external routines are called from within the program which reside in a library named "DOE.MISC.OBJLIB". The following JCL statements would compile the program and execute it.

```
//DOE JOB
//STEP1 EXEC PASCCG,
//           LKLBDSN='DOE.MISC.OBJLIB'
//PASC.SYSIN DD DSN=DOE.SEARCH.PASCAL,
//              DISP=SHR
//GO.SYSLIN DD
//            DD DSN=DOE.SORT.OBJ,
//              DISP=SHR
```

```
//PASCCL   PROC SYSOUT=*,OPTIONS=,INCLLIB='SYS1.PASCALVS.MACLIB',
//              LKLBDSN='SYS1.PASCALVS.LOAD',
//              LINKLIB='SYS1.PASCALVS.LINKLIB'
//PASC     EXEC  PGM=PASCALL,PARM='&OPTIONS'

          ... (this step is identical to the PASC step in procedure PASCC)

//PASCT    EXEC  PGM=PASCALT,PARM='&OPTIONS'

          ... (this step is identical to the PASCT step in procedure PASCC)

//*
//*   L K E D
//*
//LKED    EXEC  PGM=IEWL,PARM='LIST,MAP',
//              COND=((8,LE,PASC),(8,LE,PASCT))
//SYSLIB    DD  DSN=&LKLBDSN,DISP=SHR
//          DD  DSN=SYS1.PASCALVS.LOAD,DISP=SHR
//SYSLMOD   DD  DSN=&&GOSET(GO),UNIT=SYSDA,DISP=(,PASS),
//              SPACE=(TRK,(5,3,1))
//SYSUT1    DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSPRINT  DD  SYSOUT=&SYSOUT
//SYSLIN    DD  DSN=&&LOADSET,DISP=(OLD,DELETE)
//          DD  DDNAME=SYSIN
```

Figure 13. Cataloged procedure PASCCL

4.4.3 Compile and Link Edit (PASCCL)

In this cataloged procedure (Figure 13), a Pascal/VS source module is compiled to produce an object module and then the linkage editor is executed to produce a load module.

The linkage editor step is named LKED. The DD statement with the name SYSLIB within this step specifies the library, or libraries, from which the linkage editor will obtain appropriate modules for inclusion in the load module. The linkage editor always places the load modules it creates in the standard data set defined by the DD statement with the name SYSLMOD. This statement in the cataloged procedure specifies a new temporary library &&GOSET, in which the load module will be placed and given the member name GO.

In specifying a temporary library, it is assumed that you will execute the load module in the same job; if you want to retain the module, you must substitute your own statement for the DD statement with the name SYSLMOD.

When linking multiple modules together, you must supply an entry point. The name of the entry point may be either the name of your main program, or the name AMPXSTRT. To define an entry point, a linkage editor ENTRY control card must be processed by the linkage editor. This may be done conveniently with a DD statement named SYSIN for step LKED which references instream data:

```
//LKED.SYSIN DD *
  ENTRY AMPXSTRT
/*
```

Multiple invocations of the PASCC cataloged procedure concatenates object modules. This permits several modules to be compiled and link edited conveniently in one job. The JCL shown in Figure 14 on page 25 compiles three source modules and then link edits them to produce a single load module. Within the example, each source module is a member of a partitioned data set named

"DOE.PASCAL.SRCLIB1".

The member names are MAIN, SEG1, and SEG2. The resulting load module is to be placed in a preallocated library named "DOE.PROGRAMS.LOAD" as a member named MAIN.

```
//JOBNAME JOB (DOE),'JOHN DOE'
//STEP1 EXEC PASCC
//PASC.SYSIN DD DSN=DOE.PASCAL.SRCLIB1(MAIN),DISP=SHR
//STEP2 EXEC PASCC
//PASC.SYSIN DD DSN=DOE.PASCAL.SRCLIB1(SEG1),DISP=SHR
//STEP3 EXEC PASCCL
//PASC.SYSIN DD DSN=DOE.PASCAL.SRCLIB1(SEG2),DISP=SHR
//LKED.SYSLMOD DD DSN=DOE.PROGRAMS.LOAD(MAIN),DISP=OLD
//LKED.SYSIN DD *
  ENTRY AMPXSTRT
/*
```

Figure 14. Sample JCL to perform multiple compiles and a link edit.

```
//PASCCLG PROC SYSOUT=*,OPTIONS=,INCLLIB='SYS1.PASCALVS.MACLIB',
//              LKLBDSN='SYS1.PASCALVS.LOAD',
//              LINKLIB='SYS1.PASCALVS.LINKLIB'
//PASC     EXEC  PGM=PASCALL,PARM='&OPTIONS'

          ... (this step is identical to the PASC step in procedure PASCC)

//PASCT    EXEC  PGM=PASCALT,PARM='&OPTIONS'

          ... (this step is identical to the PASCT step in procedure PASCC)

//LKED     EXEC  PGM=IEWL,PARM='LIST,MAP',

          ... (this step is identical to the LKED step in procedure PASCCL)

//GO       EXEC  PGM=*.LKED.SYSLMOD,
//               COND=((8,LE,PASC),(8,LE,PASCT),(8,LE,LKED))
//SYSPRINT DD  SYSOUT=&SYSOUT
//OUTPUT   DD  SYSOUT=&SYSOUT,DCB=(RECFM=VBA,LRECL=133,BLKSIZE=685)
//INPUT    DD  DUMMY,DCB=(RECFM=V,LRECL=256,BLKSIZE=260)
```

Figure 15. Cataloged procedure PASCCLG

4.4.4 Compile, Link Edit, and Execute (PASCCLG)

This cataloged procedure (Figure 15) performs a compilation, invokes the linkage editor to form a load module from the resulting object module, then the load module is executed.

The first three steps of this procedure are identical to those of the PASCCL procedure. An additional fourth step (named GO) executes your program.

4.5 HOW TO ACCESS AN %INCLUDE LIBRARY

The DD statement named SYSLIB in procedure step PASC defines the libraries from which included source is to be retrieved.

When the compiler encounters an %INCLUDE statement within the source module, it will search the library or libraries specified by SYSLIB for the member named in the statement. When found, the library member becomes the input stream for the compiler. After the compiler has read the entire member, it will continue where it left off in the previous input stream.

You may specify an %INCLUDE library by means of the INCLLIB parameter of the cataloged procedures, or by overriding the SYSLIB DD statement by specifying a DD statement with the name PASC.SYSLIB.

Example

```
//JOBNAME JOB
//  EXEC PASCCG
//PASC.SYSLIB DD DSN=...,DISP=SHR
//PASC.SYSIN DD *
   ...
/*
```

4.6 HOW TO ACCESS DATA SETS

Every file variable operated upon in your program must have an associated DD statement for the GO step which executes your program. The DDname to be associated with a particular file variable in your program is normally the name of the variable itself, truncated to eight characters.

For example, the DDnames for the variables declared within the Pascal declaration below would be SYSIN, SYSPRINT, and OUTPUTFI, respectively.

```
var
  SYSIN,
  SYSPRINT: TEXT;
  OUTPUTFILE: file of
                 INTEGER;
```

The files named OUTPUT and INPUT need not be explicitly defined by you if you use the cataloged procedures. Both cataloged procedures which execute a Pascal/VS program (PASCCG and PASCCLG) contain DD statements for OUTPUT and INPUT. OUTPUT is assigned to the output class where the system messages and compiler listings are produced (SYSOUT=*). INPUT is defined as a dummy data set.

If the Pascal/VS input/output manager attempts to open a data set which has an incomplete data control block (DCB), it will assign default values to the DCB as described in "Data Set DCB Attributes" on page 39. If you prefer not to rely on the defaults, then the LRECL, BLKSIZE, and RECFM should be explicitly specified in the DCB operand of the associated DD statement for a newly created data set (that is, one whose DISP operand is set to NEW).

4.7 EXAMPLE OF A BATCH JOB

```
//JOBNAME JOB
//STEP1 EXEC PASCC,OPTIONS='NOXREF'
//PASC.SYSIN DD *
program COPYFILE;
type
  F80    = file of
              packed array[1..80] of CHAR;
var
  INFILE, OUTFILE: F80;
procedure COPY(var FIN,FOUT: F80);
  external;
begin
  RESET(INFILE);
  REWRITE(OUTFILE);
  COPY(INFILE,OUTFILE);
end.
/*
//STEP2 EXEC PASCCLG,OPTIONS='NOXREF'
//PASC.SYSIN DD *
segment IO;
type
  F80    = file of
              packed array[1..80] of CHAR;
procedure COPY(var FIN,FOUT: F80);
  entry;
begin
  while not EOF(FIN) do
    begin
      FOUT@ := FIN@;
      PUT(FOUT);
      GET(FIN)
    end
end;.
/*
//LKED.SYSIN DD *
  ENTRY COPYFILE
/*
//GO.INFILE DD *
         ...
  (data to be copied into data set goes here)
         ...
/*
//GO.OUTFILE DD DSN=P656706.TEMP.DATA,UNIT=TSOUSER,
//              DISP=(NEW,CATLG),
//              DCB=(RECFM=FB,LRECL=80,BLKSIZE=3120),
//              SPACE=(3120,(1,1))
```

Figure 16. Example of a batch job

5.0 COMPILER OPTIONS

Compile time options indicate what features are to be enabled or disabled when the compiler is invoked. The following table lists all compiler options with their abbreviated forms and their default values.

Compiler Option	Abbreviated Name	Default
CHECK/NOCHECK	---	CHECK
DEBUG/NODEBUG	---	NODEBUG
GOSTMT/NOGOSTMT	GS/NOGS	GOSTMT
LIST/NOLIST	---	NOLIST
MARGINS(m,n)	MAR(m,n)	MARGINS(1,72)
OPTIMIZE/NOOPTIMIZE	OPT/NOOPT	OPTIMIZE
SEQUENCE(m,n)/NOSEQUENCE	SEQ(m,n)/NOSEQ	SEQUENCE(73,80)
SOURCE/NOSOURCE	S/NOS	SOURCE
WARNING/NOWARNING	W/NOW	WARNING
XREF/NOXREF	X/NOX	XREF(SHORT)

5.1 CHECK/NOCHECK

If the **CHECK** option is enabled, the Pascal/VS compiler will generate inline code to perform runtime error checking. The **%CHECK** feature can be used to enable or disable particular checking code at specific locations within the source program. If **NOCHECK** is specified, all runtime checking will be suppressed and all **%CHECK** statements will be ignored. The runtime errors which may be checked are listed as follows:

CASE statements
Any **case** statement that does not contain an **otherwise** clause is checked to make sure that the selector expression has a value equal to one of the case label values.

Function routines
A call to a function routine is checked to verify that the called function returns a value.

Pointers
A reference to an object which is based upon a pointer variable is checked to make sure that the pointer does not have the value **nil**.

Subrange scalars
Variables which are declared as subrange scalars are tested when they are assigned a value to guarantee that the value lies within the declared bounds of the variable. This checking may occur when either the variable appears on the left side of an assignment statement or immediately after a routine call in which the variable was passed as a **var** parameter. (This latter case also includes a call to the READ procedure).

For the sake of efficiency, the compiler may suppress checking when it is able to determine that it is semantically unnecessary. For example, the compiler will not generate code to check the first three assignment statements below; however, the last three will be checked.

```
var
  A : -10..10;
  B : 0..20;
 ...
 A := B - 10;  (*no check*)
 B := ABS(A);  (*no check*)
 A := B DIV 2; (*no check*)
 ...
 A := B;       (*check   *)
 B := A*10;    (*check   *)
 A := -B;      (*check   *)
```

The compiler makes no explicit attempt to diagnose the use of uninitialized variables.

Subscript ranges
Subscript expressions within arrays or spaces are tested to guarantee that their values lie within the declared array or space bounds. As in the case of subrange checks, the compiler will suppress checks that are semantically unnecessary.

When a runtime checking error occurs, a diagnostic message will be sent to the file OUTPUT followed by a traceback of the routines which were active when the error occurred. See "Reading a Pascal/VS Trace Back" on page 49 for an example of a traceback due to a checking error.

5.2 DEBUG/NODEBUG

An interactive debugging facility is available to debug Pascal/VS programs. The debugger is described in "Debug - Pascal/VS Interactive Debugger" on page 53. If the option DEBUG is enabled, the compiler will produce the necessary information that Debug needs in order to operate.

The DEBUG option also implies that the GOSTMT option is active.

NODEBUG indicates that Debug cannot be used for this segment.

5.3 GOSTMT/NOGOSTMT

The GOSTMT option enables the inclusion of a statement table within the object code. The entries within this table allow the run-time environment to identify the source statement causing an execution error. This statement table also permits the interactive debugger to place breakpoints based on source statement numbers. For a description of the debugger see "Debug - Pascal/VS Interactive Debugger" on page 53.

The inclusion of the statement table does not affect the execution speed of the compiled program.

NOGOSTMT will prevent the statement table from being generated.

5.4 LIST/NOLIST

The LIST/NOLIST option controls the generation or suppression of the translator pseudo-assembler listing (see "Assembly Listing" on page 37).

Note: The NOLIST option will cause any %LIST statement within the source program to be ignored.

5.5 MARGINS(M,N)

The MARGINS(m,n) option sets the left and right margin of your program. The compiler scans each line of your program starting at column m and ending at column n. Any data outside these margin limits is ignored. The maximum right margin allowed is 80.

The specified margins must not overlap the sequence field. A specification of MARGINS(1,80) implies that the source contains no sequence numbers.[6]

The default is MARGINS(1,72).

Note: When the PASCALVS clist is being invoked under TSO, the subparameters of the MARGINS option must be enclosed in quotes. For example,

```
MARGINS('1,72')
```

5.6 OPTIMIZE/NOOPTIMIZE

The OPTIMIZE option indicates that the compiler is to generate optimized code. NOOPTIMIZE indicates that the compiler is not to optimize.

5.7 SEQ(M,N)/NOSEQ

The SEQ(m,n) option specifies which columns within the program being compiled are reserved for a sequence field. The starting column of the sequence field is m; the last column of the field is n.

The compiler does not process sequence fields; but serve only to identify lines in the source listing. If the sequence field is blank, the compiler will insert a line number in the corresponding area in the source listing.

NOSEQ indicates that there is to be no sequence field.

The default is SEQ(73,80).

NOTES:

- The sequence field must not overlap the source margins.
- When the PASCALVS clist is being invoked under TSO, the subparameters of the SEQ option must be enclosed in quotes. For example,

```
SEQ('73,80')
```

5.8 SOURCE/NOSOURCE

The SOURCE/NOSOURCE option controls the generation or suppression of the compiler source listing.

Note: The NOSOURCE option will cause any %PRINT statement within the source program to be ignored.

[6] The option **NOSEQUENCE** has the same effect.

5.9 WARNING/NOWARNING

This option controls the generation or suppression of warning messages. The NOWARNING specification will suppress warning messages from the compiler.

5.10 XREF/NOXREF

The XREF/NOXREF option controls the generation or suppression of the cross-reference portion of the source listing. (See "Cross-reference Listing" on page 35).

Either a short or long cross-reference listing can be generated. A long cross-reference listing contains all identifiers declared in the program. A short listing consists of only those identifiers which were referenced.

To specify a particular listing mode, either the word LONG or SHORT is placed after the XREF specification and enclosed within parentheses. If no such specification exists, SHORT is assumed. For example, the specification

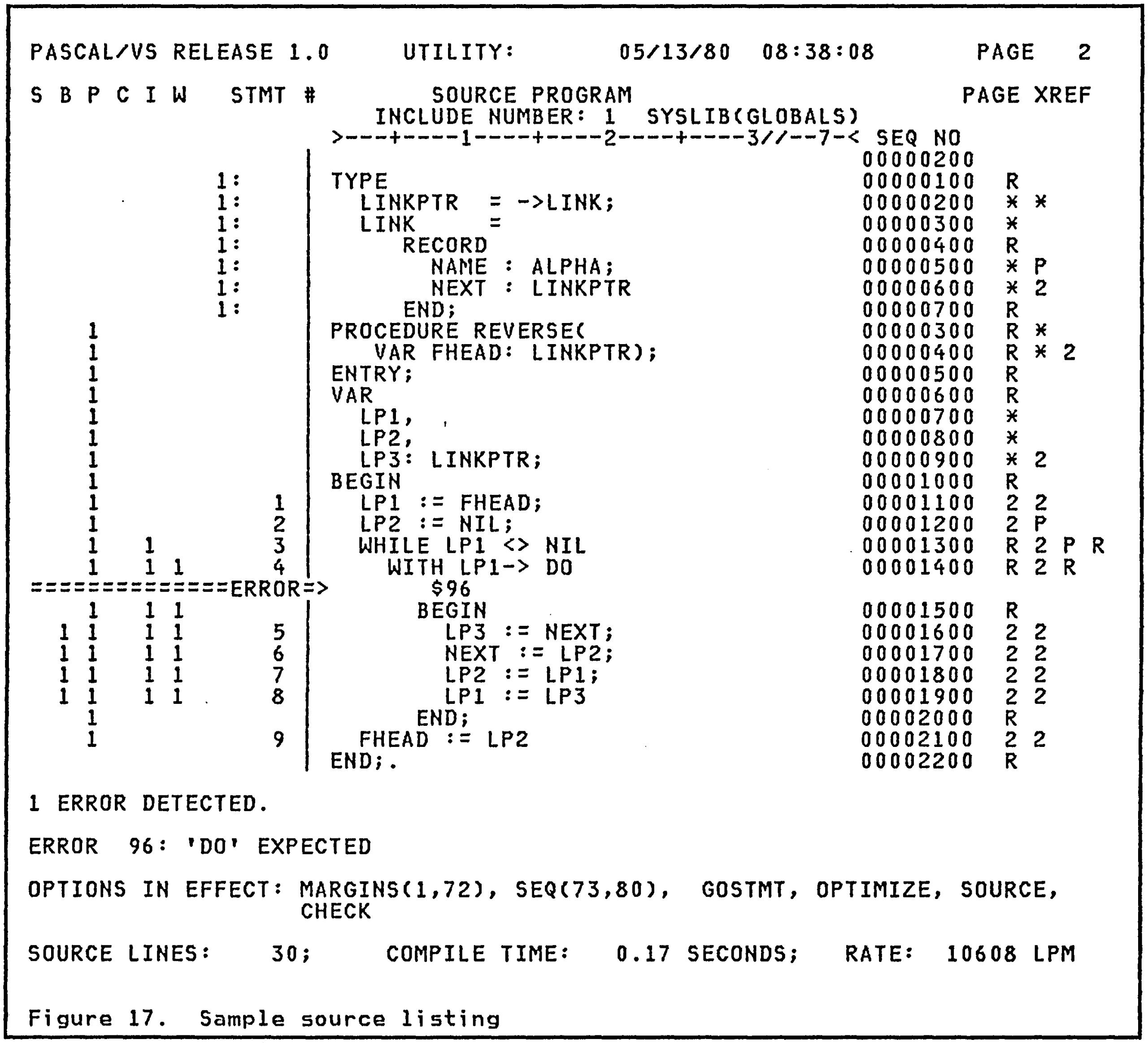
```
XREF(LONG)
```

would cause a long cross-reference table to be generated.

Note: If the PASCALVS clist is being invoked under TSO, a subparameter (SHORT or LONG) must be specified with the XREF option; there are no defaults.

6.0 HOW TO READ PASCAL/VS LISTINGS

6.1 SOURCE LISTINGS



```
PASCAL/VS RELEASE 1.0     UTILITY:       05/13/80  08:38:08      PAGE    2

S B P C I W   STMT #       SOURCE PROGRAM                     PAGE XREF
                        INCLUDE NUMBER: 1  SYSLIB(GLOBALS)
                     >---+----1----+----2----+----3//--7-< SEQ NO
                   |                                      00000200
             1:    | TYPE                                 00000100  R
             1:    |   LINKPTR  = ->LINK;                 00000200  * *
             1:    |   LINK     =                         00000300  *
             1:    |      RECORD                          00000400  R
             1:    |        NAME : ALPHA;                 00000500  * P
             1:    |        NEXT : LINKPTR                00000600  * 2
             1:    |      END;                            00000700  R
    1              | PROCEDURE REVERSE(                   00000300  R *
    1              |    VAR FHEAD: LINKPTR);              00000400  R * 2
    1              | ENTRY;                               00000500  R
    1              | VAR                                  00000600  R
    1              |   LP1,                               00000700  *
    1              |   LP2,                               00000800  *
    1              |   LP3: LINKPTR;                      00000900  * 2
    1              | BEGIN                                00001000  R
    1            1 |   LP1 := FHEAD;                      00001100  2 2
    1            2 |   LP2 := NIL;                        00001200  2 P
    1   1        3 |   WHILE LP1 <> NIL                   00001300  R 2 P R
    1   1 1      4 |     WITH LP1-> DO                    00001400  R 2 R
==============ERROR=>       $96
    1   1 1        |       BEGIN                          00001500  R
  1 1   1 1      5 |         LP3 := NEXT;                 00001600  2 2
  1 1   1 1      6 |         NEXT := LP2;                 00001700  2 2
  1 1   1 1      7 |         LP2 := LP1;                  00001800  2 2
  1 1   1 1      8 |         LP1 := LP3                   00001900  2 2
    1              |       END;                           00002000  R
    1            9 |   FHEAD := LP2                       00002100  2 2
                   | END;.                                00002200  R

1 ERROR DETECTED.

ERROR  96: 'DO' EXPECTED

OPTIONS IN EFFECT: MARGINS(1,72), SEQ(73,80),  GOSTMT, OPTIMIZE, SOURCE,
                   CHECK

SOURCE LINES:    30;     COMPILE TIME:    0.17 SECONDS;    RATE:  10608 LPM
```

Figure 17. Sample source listing

The source listing contains information about the source program including nesting information of blocks and cross reference information.

6.1.1 Page Headers

The first line of every page contains the title, if one exists. The title is set with the %TITLE statement and may be reset whenever necessary. If no title has been specified, then the line will be blank.

The second line begins with "PASCAL/VS RELEASE x". This line lists information in the following order.

1. The PROGRAM/SEGMENT name is given before a colon. This name becomes the name of the control section (CSECT) in which the generated object code will reside.
2. Following the colon may be the name of the procedure/function definition which was being compiled when the page boundary occurred.
3. The time and date of the compile.

4. The page number.

The third line contains column headings. If the source being compiled came from a library (i.e. %INCLUDE), then the last line of the heading identifies the library and member.

6.1.2 Nesting Information

The left margin contains nesting information about the program. The depth of nesting is represented by a number. The heading over this margin is:

```
S P B C I W   STMT
```

S - a '1' in this column indicates that the line contains a comment which 'S'pans across the line.

P - indicates the depth of 'P'rocedure nesting.

B - indicates the depth of 'B'EGIN block nesting.

C - indicates the nesting of 'C'onditional statements. Conditional statements are **if** and **case**.

I - indicates the nesting of 'I'terative statements. Iterative statements are **for**, **repeat** and **while**.

W - indicates the nesting of 'W'ITH statements.

STMT is the heading of a column that numbers the executable statements of each routine. If the source line orginated from an INCLUDE file, the include number and a colon (':') precede the statement number.

6.1.3 Statement Numbering

Pascal/VS numbers each executable statement according to the following rules:

- Every assignment, **if**, **for**, **while**, **case**, **with**, procedure call, and **assert** statement is given a number.
- The **until** part of a **repeat** statement is given a number.

A **begin/end** statement is not numbered because it serves only as a bracket for a sequence of statements and has no executable code associated with it. The statement numbers are given for runtime errors and to specify breakpoints in the interactive debugger (see "Debug - Pascal/VS Interactive Debugger" on page 53).

6.1.4 Page Cross Reference

The right margin contains an indicator for each identifier that appears in the associated line. The indicators have the following meanings:

- A number indicates a page number on which the corresponding identifier was declared.
- A '*' indicates that the corresponding identifier is being declared.
- A 'P' indicates that the corresponding identifier is predefined.
- A 'R' indicates that the corresponding identifier is a reserved key word.
- A '?' indicates that the corresponding identifier is either undeclared, or will be declared further on in the program. This latter occurrence arises often in pointer type definitions.

6.1.5 Error Summary

Toward the end of the listing is the error summary. It contains the diagnostic messages corresponding to the compilation errors detected in the program.

6.1.6 Option List

The option list summarizes the options that were enabled for the compilation.

6.1.7 Compilation Statistics

The compiler prints summary statistics which tell the number of lines compiled, the time required, and compilation rate in lines per minute of (virtual) CPU time.

These statistics are divided between two phases of the compiler: the syntax/semantic phase and the code generation phase. Also printed is the total time and accumulative rate for the sum of the phases.

6.2 CROSS-REFERENCE LISTING

```
                 C R O S S   R E F E R E N C E   L I S T I N G

           INCLUDE 1 CAME FROM MEMBER GLOBALS

IDENTIFIER       DEFINITION   ATTRIBUTES        <PAGE #>/<INCLUDE #>:<LINE #>

ALPHA            PREDEFINED   CLASS = TYPE, TYPE = ARRAY, LENGTH = 16
                              2/1:5

FHEAD              2/4        IN REVERSE, CLASS = VAR PARAM,
                              TYPE = POINTER, OFFSET = 144, LENGTH = 4
                              2/11        2/21

LINK               2/1:3      CLASS = TYPE, TYPE = RECORD, LENGTH = 20
                              2/1:2

LINKPTR            2/1:2      CLASS = TYPE, TYPE = POINTER, LENGTH = 4
                              2/1:6       2/4         2/9

LP1                2/7        IN REVERSE, CLASS = LOCAL VAR, TYPE = POINTER,
                              OFFSET = 148, LENGTH = 4
                              2/11        2/13        2/14        2/18
                              2/19

LP2                2/8        IN REVERSE, CLASS = LOCAL VAR, TYPE = POINTER,
                              OFFSET = 152, LENGTH = 4
                              2/12        2/17        2/18        2/21

LP3                2/9        IN REVERSE, CLASS = LOCAL VAR, TYPE = POINTER,
                              OFFSET = 156, LENGTH = 4
                              2/16        2/19

NEXT               2/1:6      IN LINK, CLASS = FIELD, TYPE = POINTER,
                              OFFSET = 16, LENGTH = 4
                              2/16        2/17

NIL              PREDEFINED   CLASS = CONSTANT, TYPE = POINTER, VALUE = 0
                              2/12        2/13

REVERSE            2/3        CLASS = PROCEDURE
```

Figure 18. Sample cross-reference listing

The cross reference listing lists alphabetically every identifier used in the program giving its attributes and both the page number and the source line number of each reference.

If the %INCLUDE facility was used, the cross reference listing will begin by listing all of the include-members by name with a reference number.

Each reference specification is of the following form:

p/ [i:] l

where **p** is the page number on which the reference occurred; **i** is the number of the include-member if the reference took place within the member; **l** is the line number within the program or include-member at which the reference occurred.

The reference immediately following the identifier is the place in the source program where the identifier was declared.

The attribute specifications have the following meaning.

IN name
If the identifier is a record field, then this attribute specifies the name of the record in which the identifier was declared; otherwise, it specifies the name of the routine in which the identifier was declared.

CLASS = class
This attribute gives the class of the identifier:

CONSTANT   declared constant

CONST PARAMETER

	pass-by-**const** parameter
DEF VAR	external **def** variable
ENTRY FUNCTION	function routine declared as an ENTRY point
ENTRY PROCEDURE	procedure routine declared as an ENTRY point
EXTERNAL FUNCTION	external function routine
EXTERNAL PROCEDURE	external procedure routine
FIELD	record field
FORMAL FUNCTION	function passed as a parameter
FORMAL PROCEDURE	procedure passed as a parameter
FORTRAN FUNCTION	external FORTRAN function
FORTRAN SUBROUTINE	external FORTRAN subroutine
FUNCTION	a user-defined or standard function
LABEL	statement label
LOCAL VAR	automatic variable
PROCEDURE	a user-defined or standard procedure
REF VAR	external **ref** variable
STATIC VAR	static variable
TYPE	type identifier
VAR PARAMETER	pass-by-**var** parameter
UNDECLARED	undeclared identifier

TYPE = type

This attributes gives the type of the identifier:

ARRAY	an array type
BOOLEAN	boolean type
CHAR	character
FILE	a file type
INTEGER	fixed point numeric
POINTER	a pointer type
REAL	floating point numeric
RECORD	a record type
SCALAR	enumerated scalar or subrange
SET	a set type
SPACE	a space type
STRING	a string type

OFFSET = n

This attribute specifies the byte offset (in decimal) within the dynamic storage area (DSA) of an automatic variable or parameter; the displacement of a record field within the associated record; or, the offset in the static area of a static variable.

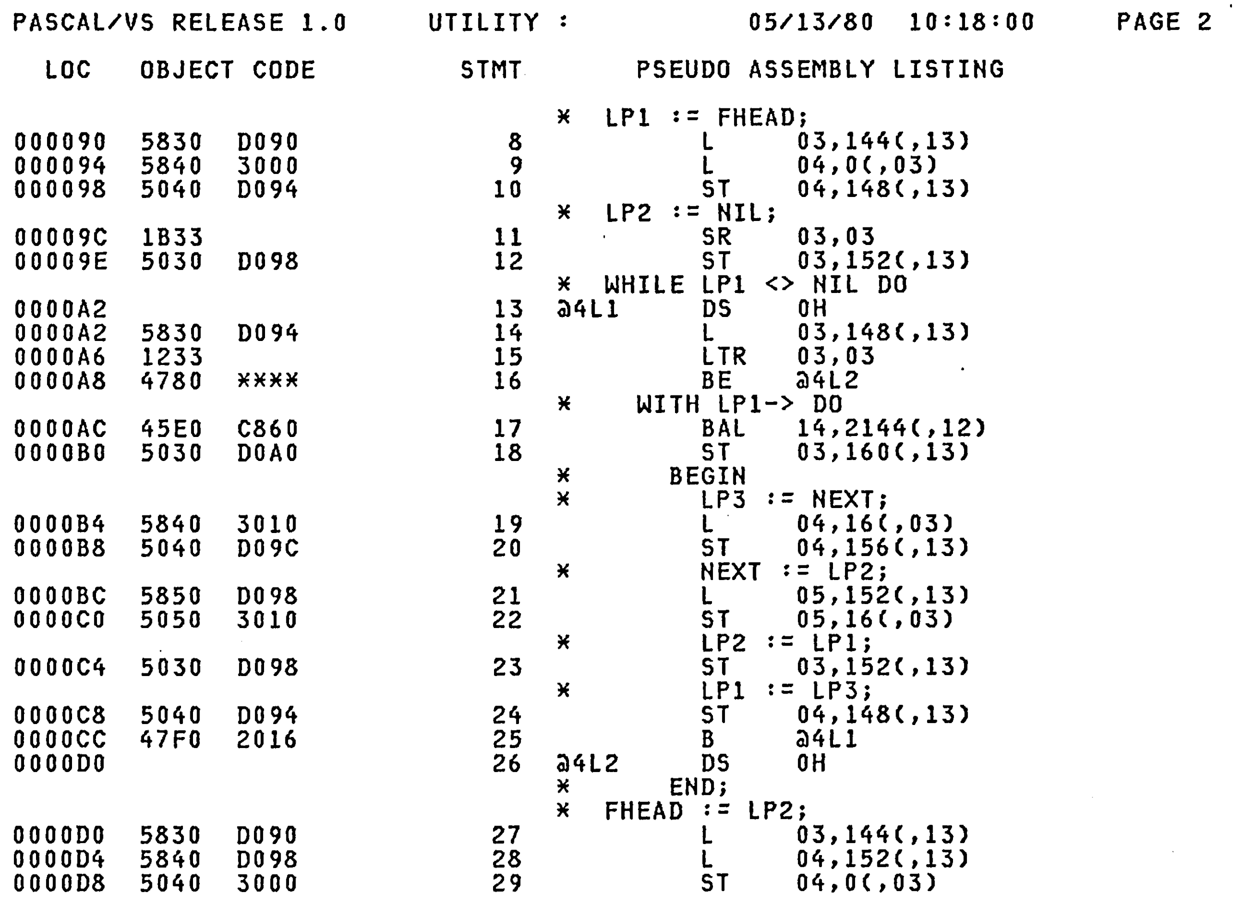
LENGTH = n

This attribute specifies the byte length of a variable or the storage required for an instance of a type.

VALUE = n

This attribute specifies the ordinal value of an integer or enumerated scalar constant.

6.3 ASSEMBLY LISTING



```
PASCAL/VS RELEASE 1.0     UTILITY :           05/13/80  10:18:00     PAGE 2

  LOC    OBJECT CODE          STMT       PSEUDO ASSEMBLY LISTING

                                      *  LP1 := FHEAD;
000090  5830  D090               8               L      03,144(,13)
000094  5840  3000               9               L      04,0(,03)
000098  5040  D094              10               ST     04,148(,13)
                                      *  LP2 := NIL;
00009C  1B33                    11               SR     03,03
00009E  5030  D098              12               ST     03,152(,13)
                                      *  WHILE LP1 <> NIL DO
0000A2                          13    @4L1       DS     0H
0000A2  5830  D094              14               L      03,148(,13)
0000A6  1233                    15               LTR    03,03
0000A8  4780  ****              16               BE     @4L2
                                      *    WITH LP1-> DO
0000AC  45E0  C860              17               BAL    14,2144(,12)
0000B0  5030  D0A0              18               ST     03,160(,13)
                                      *      BEGIN
                                      *      LP3 := NEXT;
0000B4  5840  3010              19               L      04,16(,03)
0000B8  5040  D09C              20               ST     04,156(,13)
                                      *      NEXT := LP2;
0000BC  5850  D098              21               L      05,152(,13)
0000C0  5050  3010              22               ST     05,16(,03)
                                      *      LP2 := LP1;
0000C4  5030  D098              23               ST     03,152(,13)
                                      *      LP1 := LP3;
0000C8  5040  D094              24               ST     04,148(,13)
0000CC  47F0  2016              25               B      @4L1
0000D0                          26    @4L2       DS     0H
                                      *      END;
                                      *  FHEAD := LP2;
0000D0  5830  D090              27               L      03,144(,13)
0000D4  5840  D098              28               L      04,152(,13)
0000D8  5040  3000              29               ST     04,0(,03)
```

Figure 19. Sample assembly listing

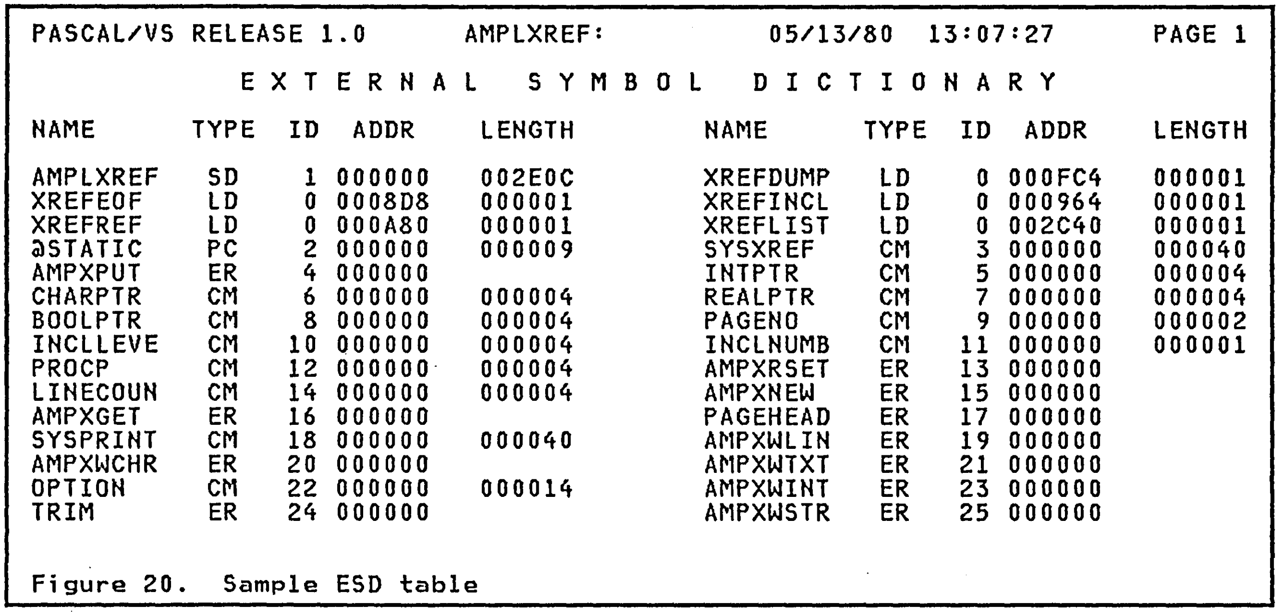
The compiler produces a pseudo assembly listing of your program if you specify the LIST option. The information provided in this listing include:

LOC
location relative to the beginning of the module in bytes (hexadecimal).

OBJECT CODE
up to 6 bytes per line of the generated text. If the line refers to a symbol or literal not yet encountered in the listing (forward reference) the base displacement format of the instruction is shown as four asterisks ('****').

PSEUDO ASSEMBLY
basic assembly language description of generated instruction.

Annotation
intermixed with the assembly instructions is the source line from which the instructions were generated. The source lines appear as comments in the listing.



```
PASCAL/VS RELEASE 1.0      AMPLXREF:          05/13/80  13:07:27      PAGE 1

                E X T E R N A L   S Y M B O L   D I C T I O N A R Y

NAME       TYPE  ID  ADDR     LENGTH          NAME       TYPE  ID  ADDR     LENGTH

AMPLXREF   SD     1 000000    002E0C          XREFDUMP   LD     0 000FC4    000001
XREFEOF    LD     0 0008D8    000001          XREFINCL   LD     0 000964    000001
XREFREF    LD     0 000A80    000001          XREFLIST   LD     0 002C40    000001
@STATIC    PC     2 000000    000009          SYSXREF    CM     3 000000    000040
AMPXPUT    ER     4 000000                    INTPTR     CM     5 000000    000004
CHARPTR    CM     6 000000    000004          REALPTR    CM     7 000000    000004
BOOLPTR    CM     8 000000    000004          PAGENO     CM     9 000000    000002
INCLLEVE   CM    10 000000    000004          INCLNUMB   CM    11 000000    000001
PROCP      CM    12 000000    000004          AMPXRSET   ER    13 000000
LINECOUN   CM    14 000000    000004          AMPXNEW    ER    15 000000
AMPXGET    ER    16 000000                    PAGEHEAD   ER    17 000000
SYSPRINT   CM    18 000000    000040          AMPXWLIN   ER    19 000000
AMPXWCHR   ER    20 000000                    AMPXWTXT   ER    21 000000
OPTION     CM    22 000000    000014          AMPXWINT   ER    23 000000
TRIM       ER    24 000000                    AMPXWSTR   ER    25 000000
```

Figure 20. Sample ESD table

6.4 EXTERNAL SYMBOL DICTIONARY

The External Symbol Dictionary (ESD) provides one entry for each name in the generated program that is an external. This information is required by the linker/loader to resolve inter-module linkages. The information in this table is:

NAME the name of the symbol.

TYPE the classification of the symbol:

SD - Symbol Definition

LD - Local Definition

ER - External Reference

CM - Common

PC - Private Code.

ID is the number provided to the loader in order to relocate address constants correctly.

ADDR is the offset in the CSECT for an LD entry.

LENGTH the size in bytes of the SD or CM entry.

The SD classification corresponds to the name of the module; the LD classifications are entry routines; ER names are external routines; CM names correspond to **def** variables. The private code section is where static variables are located.

6.5 INSTRUCTION STATISTICS

If Pascal/VS is requested to produce an assembly listing, it will also summarize the usage of 370 instructions generated by the compiler. The table is sorted by frequency of occurrence.

7.0 USING INPUT/OUTPUT FACILITIES

7.1 I/O IMPLEMENTATION

Pascal/VS employs OS access methods to implement its input/output facilities. Pascal/VS file variables are associated with a data set by means of a ddname. The Queued Sequential Access Method (QSAM) is used for sequential data sets. The Basic Partitioned Access Method (BPAM) is used for partitioned data sets (MACLIBs in CMS terminology).

7.2 DDNAME ASSOCIATION

For any identifier declared as a file variable the first eight characters of the identifier's name serves as the DDNAME of the file. As a consequence, the first eight characters of all file variables declared within a module should be unique. You must also be careful not to allow one of the first eight characters to be an underscore ('_') since this is not a valid character to appear in a DDNAME.

If you prefer, you may associate an arbitrary ddname with a file variable by explicitly specifying a ddname within the OPEN procedure (see "The OPEN Procedure" on page 46).

7.3 DATA SET DCB ATTRIBUTES

At runtime, associated with every Pascal/VS file variable is a Data Control Block (DCB) which contains information describing specific attributes of the associated data set. Among these attributes are

- the logical record length (LRECL);
- the physical block size (BLKSIZE);
- the record format (RECFM).

Pascal/VS supports only the following record formats:

F, FA, FB, FBA, V, VA, VB, VBA

Newly allocated (empty) data sets, that is, data sets intended for output might not have these attributes assigned. As far as Pascal/VS is concerned, there are two ways to specify the DCB attributes for such data sets:

- by being specified in the associated DDNAME definition (in CMS: the FILEDEF command; in TSO: the ALLOC/ATTR commands; in OS batch: the DD card);
- by being specified in the OPEN procedure (see "The OPEN Procedure" on page 46).

If any of these attributes are unassigned for a particular data set to which a Pascal/VS program will be writing, the Pascal/VS I/O manager will assign defaults according to whether the data set is being managed as a file of type "TEXT" or as a non-TEXT file.

For the case of TEXT files, if neither LRECL, BLKSIZE, nor RECFM are specified, then the following defaults will apply:

- LRECL=256
- BLKSIZE=260
- RECFM=V

For the case of non-TEXT files, if neither LRECL, BLKSIZE, nor RECFM are specified then the following defaults will apply.

- LRECL="length of file component"
- BLKSIZE=LRECL
- RECFM=F

If some of the attributes are specified and some are not then defaults will be applied using the following criteria:

- RECFM of V is preferred over F for TEXT files.
- RECFM of F is preferred over V for non-TEXT files.
- If RECFM is F then the BLKSIZE is to be equal to the LRECL or to be a multiple thereof.
- If RECFM is V then the BLKSIZE is to be at least four bytes greater than the LRECL.

7.4 TEXT FILES

Text files contain character data grouped into logical records. From a Pascal/VS language viewpoint, the logical records are lines of characters. Pascal/VS supports both fixed length and variable length record formats for text files. Characters are stored in their EBCDIC representations.

The predefined type TEXT is used to declare a text file variable in

Pascal/VS. The pointer associated with each file variable points to positions within a physical I/O buffer.

7.4.1 Opening a Text File

A closed file is opened automatically by the procedures GET and READ for input, and WRITE for output[7]. To open a file explicitly, the procedures RESET, REWRITE, INTERACTIVE, and OPEN are provided.

The procedures RESET and INTERACTIVE are used to open a file for input. RESET allocates a buffer, reads the first logical record of the file into the buffer, and positions the file pointer at the beginning of the buffer. Therefore, given a text file F, the execution of the statement 'RESET(F)' would imply that 'F->' would reference the first character of the file. If a RESET operation is performed on an open file, the file is closed and then reopened.

```
program EXAMPLE;
var
  SYSIN   : TEXT;
  C       : CHAR;
begin
  (*open SYSIN for input    *)
  RESET(SYSIN);
  (*use first char of file*)
  C := SYSIN->;
  WRITELN(C);
end.
```

Figure 21. Using RESET on a TEXT file

Since RESET performs an implicit read operation to fill a file buffer, it is not well suited for files intended to be associated with interactive input. To alleviate this problem you should use the INTERACTIVE procedure to open the file. No initial read operation is performed on files opened in this manner. The file pointer has the value NIL until the the first file operation is performed (namely GET or READ).

```
program EXAMPLE;
var
  SYSIN   : TEXT;
  DATA    : STRING(80);
begin
  (*open SYSIN for interactive *)
  (*input                      *)
  INTERACTIVE(SYSIN);
  (*prompt for response        *)
  (*read in response           *)
  WRITELN(' ENTER DATA: ');
  READLN(SYSIN,DATA);
end.
```

Figure 22. Using INTERACTIVE on a TEXT file

The procedure REWRITE is used to open a file for output. The file pointer is positioned at the beginning of an empty buffer. If the file is already open it is closed prior to being reopened.

```
program EXAMPLE;
var
  SYSPRINT : TEXT;
begin
  REWRITE(SYSPRINT);
  WRITELN(SYSPRINT,'MESSAGE');
end.
```

Figure 23. Using REWRITE on a TEXT file

7.4.2 Text File PUT

The PUT procedure, when applied to an output text file, causes the file pointer to be incremented by one character position. If, prior to the call, the number of characters in the current logical record is equal to the file's logical record length (LRECL), the file pointer will be positioned within the associated buffer to begin a new logical record.

When the file buffer is filled to capacity, the buffer is written to the associated physical file. The file pointer is then positioned to the beginning of the buffer so that it may be refilled on subsequent calls to PUT. The capacity of the buffer is equal to the file's physical block size (BLKSIZE).

To terminate a logical record before it is full requires a call to WRITELN (see "The WRITELN Procedure" on page 44).

[7] The procedure PUT does not perform an implicit open on a file. Prior to a PUT operation, the associated output buffer must contain the data to be written. If the file is not open when the PUT operation is attempted, then no output buffer exists. (The file pointer will have the value **nil**.)

```
program EXAMPLE;
var
  OUTFILE   : TEXT;
  C         : CHAR;
  ...
begin
  REWRITE(OUTFILE);
  ...
  OUTFILE-> := C;
  (*Write out value of C*)
  PUT(OUTFILE);
  ...
end.
```

Figure 24. Using PUT on a TEXT file

7.4.3 Text File GET

The GET procedure, when applied to an input text file, causes the file pointer to be incremented by one character position. If the file pointer is positioned at the last position of a logical record, the GET operation will cause the end-of-line condition to become true (see "End of Line Condition") and the file pointer will be positioned to a blank. If, prior to the call, the end-of-line condition is true, then the file pointer will be positioned to the beginning of the next logical record.

If GET is called when the file pointer is positioned at the last character position of the file, the end-of-file condition becomes true. (See "End of File Condition" on page 42).

```
program EXAMPLE;
var
  INFILE    : TEXT;
  C1,C2     : CHAR;
  ...
begin
  (*get first char of file*)
  RESET(INFILE);
  C1 := INFILE->;
  (*get second char of file*)
  GET(INFILE);
  C2 := INFILE->;
  ...
end.
```

Figure 25. Using GET on a TEXT file

7.4.4 The PAGE Procedure

The PAGE procedure causes a page eject to occur on a text output file which is to be associated with a printer (or to a disk file which will eventually be printed).

```
program EXAMPLE;
var
  PRINT:  TEXT;
begin
  ...
  (*start new page*)
  PAGE(PRINT);
  ...
end.
```

Figure 26. Using the PAGE procedure

7.4.5 End of Line Condition

The end-of-line condition occurs on a text file opened for input when the file pointer is positioned after the end of a logical record. To test for this condition, the EOLN function is used.

The end-of-line condition becomes true when GET is executed for a file positioned at the last character of a logical record, or if a call to READ consumes all of the characters of the current logical record.

The file pointer will always point to a blank character (in EBCDIC, hexadecimal 40) when the end-of-line condition occurs.

The EOLN function is only applicable to text files.

```
program EXAMPLE;
var
  SYSIN:  TEXT;
  CNT   : 0..32767;
begin
  (* compute length of first
     logical record of SYSIN *)
  RESET(SYSIN);
  CNT := 0;
  while not EOLN(SYSIN) do
    begin
      CNT := CNT + 1;
      GET(SYSIN);
    end;
  WRITELN(CNT)
end.
```

Figure 27. Using the EOLN function

7.4.6 End of File Condition

The end-of-file condition becomes true when GET is executed for a file positioned at the last character of the last logical record, or if a call to READ consumes all of the characters of the last logical record.

The file pointer will always point to a blank character (hexadecimal 40) when the end-of-file condition occurs. To test for this condition, the EOF function is used.

Any calls to GET or READ for a file for which the end-of-file condition is true will be ignored.

```
program EXAMPLE;
var
  SYSIN:  TEXT;
  CNT   : 0..32767;
begin
  (* compute number of logical
     records in file SYSIN   *)
  RESET(SYSIN);
  CNT := 0;
  while not EOF(SYSIN) do
    begin
      CNT := CNT + 1;
      READLN(SYSIN)
    end;
  WRITELN(CNT)
end.
```

Figure 28. Using the EOF function on a TEXT file

7.4.7 Text File READ

The READ procedure fetches data from a text file beginning at the current position of the file pointer. If the file pointer is not yet set, an initial GET operation is performed. This case occurs when a file is opened INTERACTIVEly.

If READ is called for a closed file, the file is opened for input by an implicit call to RESET.

When reading INTEGER or REAL data via the READ procedure, and no length field is specified, all blanks preceding the data are skipped. In addition, logical record boundaries will be skipped. If the end-of-file condition should occur before a nonblank character is detected, the integer value 0 or the real value 0.0 will be returned.

Integer data begins with an optional sign ('+' or '-') followed by all digits up to, but not including, the first non-digit or up to the end of the logical record.

For example, given an input file positioned at the beginning of a logical record with the following contents:

```
95123SAN JOSE,CA
```

an integer read operation would bring in the value 95123. After the read, the file pointer would be positioned to the first 'S' character.

Real data begins with an optional sign ('+' or '-') and includes all of the following nonblank characters until one is detected that does not conform to the syntax of a real number.

For example, given an input file positioned at the beginning of a logical record with the following contents:

```
3.14159/2
```

a floating point read operation would bring in the floating point value 3.14159. After the read, the file pointer would be positioned to the '/' character.

The length field is the expression indicated in the following sample statement:

```
READ(file, variable : length_field);
```

If a length field value is specified, as many characters as are indicated by the value will be consumed by the read operation. The variable will be assigned from the beginning of the field. If the field is not exhausted after the variable has been assigned the data, the rest of the field will be skipped.

```
program EXAMPLE;
var
  ZIP,
  MAN    :  INTEGER;
  BALANCE: REAL;
begin
  READ(ZIP:5,MAN:6,BALANCE:9);
  WRITELN('ZIP = ',ZIP);
  WRITELN('MAN = ',MAN);
  WRITELN('BALANCE = ',BALANCE:8:2)
end.
```

Given the following input stream from file INPUT:

```
951239999991000.00JUNK
```

This program produces the following on file OUTPUT:

```
ZIP =        95123
MAN =       999999
BALANCE =  1000.00
```

Immediately after the READ statement was executed, file INPUT was positioned to the 'N' character.

Figure 29. Using READ with length qualifiers.

When reading data into variables declared as **packed array of** CHAR or STRING, data is read until one of the following three conditions occurs:

- the variable is filled to its declared capacity;
- an end-of-line condition is detected;
- the length field (if specified) is exhausted.

The length of a STRING variable will be set to the number of characters read. A variable declared as **packed array of** CHAR will be padded if necessary with blanks up to its declared length.

```
program DOREAD;
var
  INFILE   : TEXT;
  R        : array[1..10] of
record
               NAME: STRING(25);
               AGE : 0..99;
               WEIGHT: REAL
end;
  I         : 1..10;
begin
  RESET(INFILE);
  for I := 1 to 10 do
    with R[I] do
      begin
        READ(INFILE,NAME,AGE);
        READ(INFILE,WEIGHT);
        READLN(INFILE)
      end;
end.
```

Figure 30. Using READ on TEXT files.

7.4.8 The READLN Procedure

The READLN procedure is applicable only to text files. It causes the characters between the file pointer position and the end of the logical record to be skipped.

In the case of text files opened with the INTERACTIVE attribute, the file pointer is positioned after the end of the logical record and the end-of-line condition is set to true. For non-INTERACTIVE files, the file pointer is positioned at the beginning of the next logical record (unless, of course, the end-of-file condition occurs).

If the end-of-line condition is true for an INTERACTIVE file prior to a call to READLN and the condition was not the result of a previous call to READLN, then the call is ignored. Two calls to READLN in succession will cause the following logical record to be skipped in its entirety.

If READLN is called for a closed file, the file is opened implicitly for input without the INTERACTIVE attribute.

```
program COPY;
var
  INFILE,
  OUTFILE : TEXT;
  BUF     : STRING(100);
begin
  RESET(INFILE);
  REWRITE(OUTFILE);
  while not EOF(INFILE) do
    begin
      READ(INFILE,BUF);
      WRITELN(OUTFILE,BUF);
      (*ignore characters after
        column 100 in each line *)
      READLN(INFILE)
    end
end.
```

Figure 31. Using the procedure READLN

7.4.9 Text File WRITE

The WRITE procedure outputs data to a text file beginning at the current position of the file pointer. If WRITE is called for a closed file, the file is opened implicitly for output.

If during a call to WRITE, the length of the logical record being produced becomes equal to the logical record length (LRECL) of the text file, the record is completed and the remaining data is placed on a new record.

```
program DOWRITE;
var
  OUTFILE : TEXT;
  R       : array[1..10] of
              record
                NAME: STRING(25);
                AGE : 0..99;
                WEIGHT: REAL
              end;
  I        : 1..10;
begin
  REWRITE(OUTFILE);
  ...
for I := 1 to 10 do
    with R[I] do
      begin
        WRITE(OUTFILE,NAME,'  ');
        WRITE(OUTFILE,AGE:3,'  ');
        WRITE(OUTFILE,WEIGHT:3:0);
        WRITELN(OUTFILE)
      end;
end.
```

Figure 32. Using WRITE on TEXT files

7.4.10 The WRITELN Procedure

The WRITELN procedure is applicable only to text files intended for output. It causes the current logical record being produced to be completed so that the next output operation will begin a new logical record.

If the record format of the file is fixed (RECFM=F), WRITELN will fill the remainder of the current record with blanks. For variable length records (RECFM=V), the record length is set to the number of bytes currently occupied by the record.

If WRITELN is called for a closed file, the file is opened implicitly for output.

```
program DOUBLESPACE;
var
  FILEIN,
  FILEOUT : TEXT;
  BUF     : STRING;
begin
  REWRITE(FILEOUT);
  RESET(FILEIN);
  while not EOF(FILEIN) do
    begin
      READLN(FILEIN,BUF);
      WRITELN(FILEOUT,BUF);
      (*insert blank line *)
      WRITELN(FILEOUT)
    end;
end.
```

Figure 33. Using the WRITELN procedure

7.5 RECORD FILES

All non-TEXT files in Pascal/VS are record files by definition. Input and output operations on record files are done on a logical record basis instead of on a character basis.

The logical record length (LRECL) of a file must be at least large enough to contain the file's base component; otherwise, an execution time error will occur when the file is opened. For example, a file variable declared as '**file of** INTEGER' will require the associated physical file to have a logical record length of at least 4 bytes.

If a file has fixed length records (RECFM=F) and the logical record length is larger than necessary to contain the files component type, then the extra space in each logical record is wasted.

7.5.1 Opening a Record File

A closed file is opened automatically when the first operation is performed on it. The procedures GET and READ will open it for input; PUT and WRITE will open it for output. To open a file explicitly, the procedures RESET, REWRITE, and OPEN are provided.

The procedure RESET is used to open a file for input. This procedure allocates a buffer, reads the first logical record of the file into the buffer, and positions the file pointer at the beginning of the buffer. Therefore, given a record file F, the execution of the statement 'RESET(F)' would imply that the term 'F->' would reference the first component of the file. If a RESET operation is performed on an open file, the file is closed and then reopened.

The procedure REWRITE is used to open a file for output. The file pointer is positioned at the beginning of an empty buffer. If the file is already open it is closed prior to being reopened.

7.5.2 Record File PUT

The PUT procedure causes the file record that was assigned to the output buffer via the file pointer to be effectively written to the associated physical file. Each call to PUT for the case of record files produces one logical record.

```
program EXAMPLE;
var
  F : file of
        record
          NAME : STRING(25);
          AGE  : 0..99;
          WEIGHT: REAL;
          SEX  : (MALE,FEMALE)
        end;
begin
  REWRITE(F);
  F->.NAME   := 'John F. Doe';
  F->.AGE    := 36;
  F->.WEIGHT := 160.0;
  F->.SEX    := MALE;
  PUT(F);
  ...
end.
```

Figure 34. Using PUT on record files

7.5.3 Record File GET

The GET procedure causes the next sequential file record to be placed in the input buffer referenced by the file pointer. Each call to GET for the case of record files reads one logical record.

```
program EXAMPLE;
var
  F : file of
        record
          NAME : STRING(25);
          AGE  : 0..99;
          WEIGHT: REAL;
          SEX  : (MALE,FEMALE)
        end;
begin
  RESET(F);
  while not EOF(F) do
    begin
      WRITE(' Name : ',
             F->.NAME);
      WRITE(' Age  : ',
             F->.AGE:3);
      ...
      WRITELN;
      GET(F)
    end
end.
```

Figure 35. Using GET on record files

7.5.4 End of File Condition

The end-of-file condition occurs when a call to GET or READ is attempted on a record file (open for input) when no more logical records remain in the file. The function EOF is used to test this condition.

7.5.5 Record File READ

As documented in the language manual, the statement

```
    READ(F,V)
```

is equivalent to

```
  begin
    V := F->;
    GET(F)
  end
```

where F and V are declared as follows:

```
  var F: file of t;
      V: t;
```

If file F is not open when READ is called, it will be opened implicitly for input.

7.5.6 Record File WRITE

As documented in the language manual, the statement

```
  WRITE(F,V)
```

is equivalent to

```
  begin
    F-> := V;
    PUT(F)
  end
```

where F and V are declared as follows:

```
  var F: file of t;
      V: t;
```

If file F is not open when WRITE is called, it will be opened implicitly for output.

```
program EXAMPLE;
type
  REC = record
          NAME : STRING(25);
          AGE  : 0..99;
          SEX  : (MALE,FEMALE)
        end;
var
  INFILE,
  OUTFILE:
      file of REC;
  BUFFER : REC;
begin
  RESET(INFILE);
  REWRITE(OUTFILE);
  while not EOF(INFILE) do
    begin
      READ(INFILE,BUFFER);
      WRITE(OUTFILE,BUFFER)
    end
end.
```

Figure 36. Using READ and WRITE on record files.

7.6 CLOSING A FILE

All files which are declared in the body of a routine are closed implicitly when the routine returns to its invoker. All files which are open when the program terminates, whether normally or abnormally, will be closed automatically by the Pascal/VS runtime environment.

If the procedures RESET, REWRITE, or OPEN are applied to an open file, the file is closed prior to being reopened.

The procedure CLOSE is provided to close a file explicitly. CLOSE is predeclared as follows:

```
  procedure CLOSE(
       var F      : filetype);
    EXTERNAL;
```

7.7 THE OPEN PROCEDURE

The OPEN procedure is a generalized form of the procedures RESET and REWRITE. OPEN is predeclared in the following fashion:

```
  procedure OPEN(
       var F      : filetype;
       const OPTIONS: STRING);
    EXTERNAL;
```

The second parameter of the OPEN procedure is a string expression. This string contains a list of options which are read at execution time. These options determine how the file is to be opened and what attributes it is to have.

The data in the string parameter has the syntax shown in the following figure:

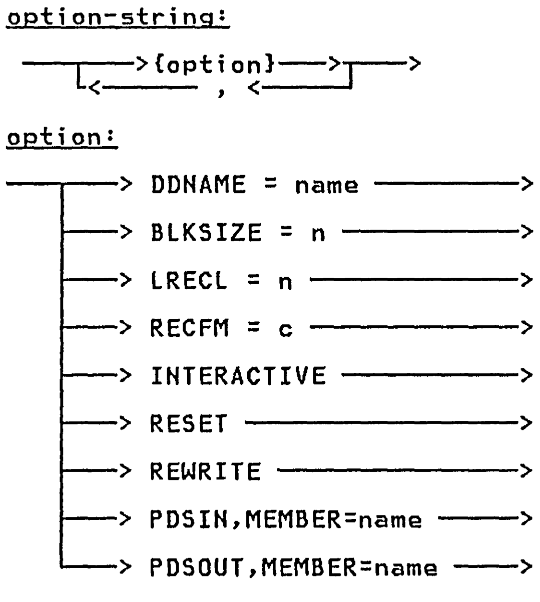


Figure 37. Syntax of string parameter of OPEN

The options RESET, REWRITE, INTERACTIVE, PDSIN, and PDSOUT are mutually exclusive. If none of these options appear in the option string, REWRITE will be assumed by default.

The following is a description of each option.

DDNAME=name

This attribute signifies that the physical file to be associated with the file variable has the ddname indicated by "name". This new ddname will remain associated with the file variable even if the file is closed and then re-opened. It can only be changed by another call to OPEN with the **DDNAME** attribute specified.

If this option is not specified, then the ddname to be associated with the file is derived from the first eight characters of the file variable name (first parameter of OPEN).

BLKSIZE=n

This attribute is used to specify a physical block size to be associated with an output file. This value (indicated by "**n**") will override a BLKSIZE specification on the ddname definition.

LRECL=n

This attribute is used to specify a logical record length to be associated with an output file. This value (indicated by "**n**") will override a LRECL specification on the ddname definition.

RECFM=c

This attribute is used to specify a record format to be associated with an output file. The only valid record formats that may be specified are

F, FB, FA, FBA, V, VB, VA, VBA

This specification (indicated by "**c**") will override a RECFM specification on the ddname definition.

INTERACTIVE

This attribute indicates that the file is to be opened for input as an interactive file. See "Opening a Text File" on page 40 for a description of interactive files.

RESET

This attribute indicates that the file is to be opened for input. A call to OPEN with this attribute performs the same function as a call to the procedure RESET.

REWRITE

This attribute indicates that the file is to be opened for output. A call to OPEN with this attribute performs the same function as a call to the procedure REWRITE.

PDSIN,MEMBER=name

PDSOUT,MEMBER=name

These attributes indicate that the file to be opened is an OS partitioned data set (PDS). The member to be accessed is indicated by "name". **PDSIN** indicates that the member is to be opened for input; **PDSOUT** indicates that it is to be opened for output. These two operations perform in the same manner as the corresponding RESET and REWRITE operations.

```
program EXAMPLE;
var
  PDS    : TEXT;
  MEMBER : STRING(8);
  BUF    : packed array[1..80] of CHAR;
begin
  OPEN(INPUT,'INTERACTIVE');            (*open INPUT for interactive *)
                                        (* input.                    *)
  READLN(MEMBER);                       (*read 1st member name       *)
  while not EOF(INPUT) do               (*loop until no more members *)
    begin                               (*open member for input      *)
      OPEN(PDS,'DDNAME=SYSLIB,PDSIN,MEMBER=' || MEMBER);
      while not EOF(PDS) do             (*copy each line of the      *)
        begin
          READLN(PDS,BUF);              (*  member to file OUTPUT    *)
          WRITELN(BUF);
        end;
      READLN(MEMBER)                    (*read next member name      *)
    end
end.
```

Figure 38. Using the OPEN procedure

7.8 PDS ACCESS IN A CMS ENVIRONMENT

In a CMS environment, members of MACLIBs may be accessed as partitioned data sets via the OS simulation facilities. A ddname is assigned to the MACLIB file with the FILEDEF command; the file name of the maclib must then appear in a "GLOBAL MACLIB" command.

For example, in order to access the file "MYLIB MACLIB A" as a partitioned data set with ddname "LIB" from a Pascal/VS program, the following commands would be executed prior to executing the program.

```
FILEDEF LIB DISK MYLIB MACLIB A
GLOBAL MACLIB MYLIB
```

Two or more MACLIBs may be accessed as though they were concatenated by using the CONCAT option of the FILEDEF command. For example, in order to access the MACLIBs "M1", "M2", and "M3" as a concatenated partitioned data set with ddname "LIB", the following commands would be executed prior to executing the Pascal/VS program.

```
FILEDEF LIB DISK M1 MACLIB A
FILEDEF LIB DISK M2 MACLIB A (CONCAT
FILEDEF LIB DISK M3 MACLIB A (CONCAT
GLOBAL MACLIB M1 M2 M3
```

8.1 READING A PASCAL/VS TRACE BACK

The Pascal/VS trace facility provides useful information while debugging programs. It gives you a list of all of the routines in the procedure chain.

For each routine the following information is given.

- The name of the routine.

- The statement number of the last statement to be executed in the routine (i.e. the statement number of the call to the next routine in the chain).

- The address in storage where the generated code for the statement begins.

- The name of the module in which the routine is declared.

The trace routine may be invoked in four different ways. You may invoke trace by placing in your source program a call to the pre-defined routine called TRACE. An example is given in Figure 39 on page 50. In the example starting at the bottom we see that Pascal/VS called the user's main program in the module named HASHASEG. Statement 24 of the main program contains the call to READ_ID, statement 3 of READ_ID contains the call to SEARCH_ID, and so on.

A trace will be produced when a program error occurs. An example is given in Figure 40 on page 50. There is an error message indicating a fixed point overflow. The traceback tells us the routine and the statement number where the error occurred. Looking at the trace we see that the error occurred at statement 3 in routine FACTORIAL on the third recursive call.

A trace will be produced when a checking error occurs. A checking error occurs when code produced by the compiler detects an invalid condition such as a subscript range error. (See "CHECK/NOCHECK" on page 29 for a description of compiler generated checks.) Figure 41 on page 50 is an example of a traceback that occurred from a checking error. The first line of the trace identifies the particular checking error that occurred. Looking at the trace we see that the error occurred at statement 4 in routine TRANSLATE.

A trace will be produced when an I/O error occurs. Figure 42 on page 50 is an example of this. In this case, statement 3 of routine INITIALIZE attempted to open a file for which no DDNAME definition existed.

Due to optimization performed by the compiler, the code which tests for an error condition may be moved back several statements. Thus, when a runtime error occurs, the statement number indicated in the traceback might be slightly less than the number of the statement from which the error was generated.

```
               TRACE BACK OF ROUTINE CALLS
ROUTINE             STMT AT ADDRESS IN MODULE
TRACE                 4    '02028C'X   AMPXSENV
HASHKEY               9    '02018C'X   HASHCSEG
GET_HASH_PTR          2    '021208'X   HASHBSEG
SEARCH_ID             9    '0213C8'X   HASHBSEG
READ_ID               3    '021550'X   HASHBSEG
<MAIN PROGRAM>       24    '020278'X   HASHASEG
PASCAL/VS             0    '02048C'X
```

Figure 39. Trace called by a user program

```
PROGRAM ERROR: FIXED POINT OVERFLOW
               TRACE BACK OF ROUTINE CALLS
ROUTINE             STMT AT ADDRESS IN MODULE
FACTORIAL             3    '02014C'X   TEST
FACTORIAL             3    '02014C'X   TEST
FACTORIAL             3    '02014C'X   TEST
<MAIN PROGRAM>       17    '020298'X   TEST
PASCAL/VS             0    '02048C'X
```

Figure 40. Trace call due to program error

```
CHECKING ERROR: HIGH BOUND
               TRACE BACK OF ROUTINE CALLS
ROUTINE             STMT AT ADDRESS IN MODULE
TRANSLATE             4    '020154'X   CONVERT
TO_ASCII             10    '02024C'X   CONVERT
<MAIN PROGRAM>       17    '020338'X   CONVERT
PASCAL/VS             0    '02048C'X
```

Figure 41. Trace call due to checking error

```
AMPX001I File could not be opened: SYSIN
               TRACE BACK OF ROUTINE CALLS
ROUTINE             STMT AT ADDRESS IN MODULE
INITIALIZE            3    '020154'X   COPY
<MAIN PROGRAM>        2    '020218'X   COPY
PASCAL/VS             0    '02048C'X
```

Figure 42. Trace call due to I/O error

8.2 RUN TIME CHECKING ERRORS

The following is a list of the possible checking errors that may occur in a Pascal/VS program at run time.

LOW BOUND
Either a subscript or a subrange variable is being assigned a value less than the lower bound of the allowed range.

HIGH BOUND
Either a subscript or a subrange variable is being assigned a value greater than the upper bound of the allowed range.

NIL POINTER
an attempt was made to reference a variable from a pointer using the value NIL.

CASE ERROR
a case expression has a value other than any of the declared case labels and there is no **otherwise** clause.

STRING CONCATENATION
the concatenation of two strings results in a string greater than 255 characters in length.

STRING TRUNCATION
there was an attempt to assign to a string a value which has more characters than the maximum length of the string.

ASSERTION FAILED
an **assert** statement was executed in which its associated boolean expression evaluated to the value FALSE.

8.3 SYMBOLIC VARIABLE DUMP

When a program error or checking error occurs, a symbolic dump of all variables which are local to the routine in which the error occurred may be produced. This dump will be produced if two conditions are met:

- The source module containing the code from which the error occurred was compiled with the **DEBUG** option.
- The Pascal/VS debug library was included in the generation of the associated load module.

The dump is written to file OUTPUT.

9.0 DEBUG - PASCAL/VS INTERACTIVE DEBUGGER

Debug is a tool that allows programmers to quickly debug Pascal/VS programs without having to write debug statements directly into their source code. Basic functions include tracing program execution, viewing the runtime values of program variables, breaking at intermediate points of execution, and displaying statement frequency counting information. The programmer uses Pascal/VS source names to reference statements and data.

In order to use Debug, you must follow these four steps:

- Compile the module to be debugged with the DEBUG option. Modules that have been compiled with the DEBUG option can be linked with modules that have not been compiled with the DEBUG option.
- When link editing your program, include the debug library. (It must be located ahead of the runtime library in search order).[8]
- Ddname INPUT must be allocated to your terminal, or to the data set from which Debug commands are to be read. Likewise, the ddname OUTPUT must be allocated to your terminal.
- When executing the load module, specify 'DEBUG/' as a parameter. This will cause the debug environment to become active, and, if INPUT has been allocated to your terminal, you will be immediately prompted for a Debug command. In the Debug environment the user may issue Debug commands and examine variables for those modules which were compiled with the DEBUG option.

9.1 QUALIFICATION

A qualification consists of a module name and a routine name. Debug uses the current qualification as the default to retrieve information for commands. The current qualification consists of the name of the routine and associated source module which was last interrupted when the debugger gained control.

At the start of a Debug session, the current qualification is the name of the module containing the main program, and the main program itself.

9.2 COMMANDS

This section describes the commands that a user may issue with the Debug facility. Every command may be abbreviated to one letter if desired except the **QUIT** and **CLEAR** commands which have no abbreviation. Square brackets ('[' and ']') are used in the command description to indicate optional parts of the command.

[8] Under CMS, the debug library is included if the **DEBUG** option is specified when invoking PASCMOD. (see "How to Build a Load Module" on page 6.)

Under TSO, the debug library is included by specifying the **DEBUG** keyword operand when invoking the PASCMOD clist. (see "How to Build a Load Module" on page 12.)

9.2.1 BREAK Command

Command Format:

BREAK [[module/] routine/] stmtno

Minimum Abbreviation:

B

Where:

module is the name of a Pascal/VS module.
routine is the name of a procedure or function in the module.
stmtno is a number of a statement in the designated routine.

This command causes a breakpoint to be set at the indicated statement. The program is stopped before the statement is executed.

The module and/or routine may be omitted in which case the defaults are taken from the current qualification. **stmtno** is the number of the statement on which to stop in the specified routine of the specified module. The statement numbers are found on the source listing.

A maximum of 8 breakpoints may be set at any one time.

9.2.2 CLEAR Command

Command Format:

CLEAR

Minimum Abbreviation:

CLEAR

There are no operands.

The CLEAR command is used to remove all breakpoints.

9.2.3 CMS Command

Command Format:

CMS

Minimum Abbreviation:

C

There are no operands.

This command activates the CMS subset mode. If the program is not being run under CMS, the command is ignored.

9.2.4 DISPLAY Command

Command Format:

DISPLAY

Minimum Abbreviation:

D

The DISPLAY command is used to display information about the current Debug session at the user's terminal. The information displayed is:

- the current qualification,
- where the user's program will resume execution upon the GO command,
- the current status of Counts,
- the current status of Tracing.

9.2.5 DISPLAY BREAKS Command

Command Format:

DISPLAY BREAKS

Minimum Abbreviation:

D B

There are no operands.

The DISPLAY BREAKS command is used to produce a list of all breakpoints which are currently set.

9.2.6 DISPLAY EQUATES Command

Command Format:

DISPLAY EQUATES

Minimum Abbreviation:

D E

There are no operands.

The DISPLAY EQUATE command is used to produce a list of all equate symbols and their current definitions.

9.2.7 EQUATE Command

Command Format:

EQUATE identifier [data]

Minimum Abbreviation:

E identifier [data]

Where:

identifier is a Pascal/VS identifier.
data is a command which the identifier is to represent.

This command causes the data to replace the identifier whenever the identifier is first token in a command.

Examples

```
equate x ,r->.b[2]->
eq     y break procx/4
eq     z
```

The first example demonstrates how a user may examine a variable without having to retype a long string every time. The next example demonstrates a way to develop a synonym for a command. The third example shows how to remove an equate.

9.2.8 GO Command

Command Format:

GO

Minimum Abbreviation:

G

There are no operands.

This command causes the program to either start or resume executing. The program will continue to execute until one of the following events occurs:

- breakpoint
- program error
- normal program exit

A breakpoint or program error will return the user to the Debug environment.

9.2.9 Help Command

Command Format:

?

Minimum Abbreviation:

?

There are no operands.

The Help command lists all Debug commands.

9.2.10 LISTVARS Command

Command Format:

LISTVARS

Minimum Abbreviation:

L

There are no operands.

This command displays the values of all variables which are local to the currently active routine.

9.2.11 Qualification Command

Command Format:

QUAL [module /] [routine]

Minimum Abbreviation:

Q [module /] [routine]

Where:

module is the name of a Pascal/VS module.
routine is the name of a procedure or function in the module.

If the user does not specify a module and/or a routine name the defaults are taken from the current qualification. The defaults are applied as follows:

- the module name defaults to the current qualification.
- the routine defaults to the main program if the associated module is a program module, or to the outermost lexical level if the module is a segment module.

The lexical scope rules of Pascal are applied when viewing variables. The current qualification provides the basis on which program names are resolved. If there is no activation of the routine available (no invocations) the user may not display local variables for that routine.

Qualification may be changed at any time during a Debug session. When a breakpoint is encountered, the qualification is automatically set to the module and the routine in which the breakpoint was set.

9.2.12 QUIT Command

Command Format:

QUIT

Minimum Abbreviation:

QUIT

There are no operands.

This command causes the program to end. It is similar to a normal program exit. The user is returned to the operating system.

9.2.13 RESET Command

Command Format:

```
RESET [[module/] routine/] stmtno
```

Minimum Abbreviation:

```
R [[module/] routine/] stmtno
```

Where:

module is the name of a Pascal/VS module.
routine is the name of a procedure or function in the module.
stmtno is a number of a statement in the designated routine.

The RESET command is used to remove a breakpoint. The defaults are the same as the BREAK command.

9.2.14 SET ATTR Command

Command Format:

```
         [ ON  ]
SET ATTR [     ]
         [ OFF ]
```

Minimum Abbreviation:

```
    [ ON  ]
S A [     ]
    [ OFF ]
```

The SET ATTR command is used to set the default way in which variables are viewed. The ON parameter specifies that variable attribute information will be displayed by default. The OFF parameter specifies that variable attribute information will not be displayed by default. The default may be overridden on the variable viewing command.

9.2.15 SET COUNT Command

Command Format:

```
SET COUNT [ ON  ]
          [ OFF ]
```

Minimum Abbreviation:

```
S C [ ON  ]
    [ OFF ]
```

The SET COUNT command is used to initiate and terminate statement counting. Statement counting is used to produce a summary of the number of times every statement is executed during program execution. The summary is produced at the end of program execution and is written to the standard file OUTPUT. Statement counting may also be initiated with the runtime COUNT option.

9.2.16 SET TRACE Command

Command Format:

```
SET TRACE [ ON  ]
          [ OFF ]
```

Minimum Abbreviation:

```
S T [ ON  ]
    [ OFF ]
```

The SET TRACE command is used to either activate or deactivate program tracing. Program tracing provides the user with a list of every statement executed in the the program. This is useful for following the execution flow during execution.

9.2.17 TRACE Command

Command Format:

TRACE

Minimum Abbreviation:

T

This command has no operands.

The TRACE command is used to produce a routine trace at the user's terminal. The procedures on the current invocation chain are listed along with the most recently executed statement in each.

9.2.18 Viewing Variables

Command Format:

, variable   [(option [)]]

Where:

variable is a Pascal variable. See the chapter entitled "Variables" in the Pascal/VS Reference Manual for the syntax of a variable.
option is either ATTR or NOATTR.

This command allows the user to obtain the contents of a variable during program execution.

The static scope rules that apply to the current qualification are applied to the specified variable. If the variable is found to be a valid reference, then its value is displayed. If the name cannot be resolved within the current qualification, the user is informed that the name is not found. If the name resolves to an automatic variable for which no activation currently exists the user is informed that the variable cannot be displayed.

As can be seen from the following examples, array elements, record fields, and dynamic variables may all be viewed. Variables are formatted according to their data type. Entire records, arrays and spaces are displayed as a hexadecimal dump. The user may view an array slice by specifying fewer indices than the declared dimension of the array. The missing indices must be the rightmost ones.

The options ATTR or NOATTR can follow a left parenthesis. The default is taken from the SET ATTR command. The initial default is NOATTR. If the user gives ATTR as an option, attributes of the variable are displayed along with the value of the variable. The attributes are the data type, memory class, length if relevant, and the routine where the variable was declared.

Note: a subscripting expression may only be a variable or constant; that is, it may contain no operators. Thus, such a reference as

```
,a[b->[j]]
```

is valid (at least syntactically), but the reference

```
,a[i+3]
```

is not a valid reference because the subscripting expression is not a variable or constant.

Examples

```
,a
,p->
,p->.b
,b[1,x].int (ATTR
,p->[x,y].b->.a[1]
```

9.2.19 Viewing Memory

Command Format:

, hex-string [: length]

Where:

hex-string is a number in hexadecimal notation.
length is an integer.

This command is used to display the contents of a specific memory location. Memory beginning at the byte specified by the hex string is dumped for the number of bytes specified by the length field. If the length is not specified memory is dumped for 16 bytes. The dump is in both hex and character formats.

The hex string must be an hexadecimal number surrounded by single quotes and followed by an 'x' (eg. '35D05'X). The length is specified in decimal.

Examples

```
,'20000'X
,'46cf0'X : 100
```

9.2.20 WALK Command

Command Format:

WALK

Minimum Abbreviation:

W

There are no operands.

This command causes the program to either start executing or resume executing. The program execution will continue for exactly one statement and then the user will be returned to Debug. This command is useful for single stepping through a section of code.

9.3 DEBUG TERMINAL SESSION

```
                program MYPROG;
                type
                  R1PTR = ->R1;
                  R1 = record
                            A : STRING(12);
                            B : INTEGER;
                            X : REAL;
                            S : set of 1..31;
                            end;
                  REC2 = record
                            INT : INTEGER;
                         end;
                  COLOR = (RED, ORANGE, YELLOW, GREEN, BLUE);

                def
                  SPAC: array[0..9] of INTEGER;

                static
                  ARR : array[1..8,1..4,1..2] of REC2;

                var
                  I : 1..8;
                  J : 1..4;
                  K : 1..2;
                  C : CHAR;
                  RP : R1PTR;
                  HUE : COLOR;


                begin
 1              C := 'A';
 2              HUE := GREEN;
 3              for I := 1 to 8 do
 4                for J := 1 to 4 do
 5                  for K := 1 to 2 do
 6                    ARR[I,J,K].INT := I + J + K;
 7              for I := 0 to 9 do
 8                SPAC[I] := I;
 9              NEW(RP);
 10             with RP-> do
                  begin
 11               A := 'NEW REC';
 12               B := 3;
 13               X := 4.5;
                  end;
 14             WRITELN('END OF PROGRAM');
                end;
```

Figure 43. Sample program for Debug session

The following series of figures is a sample Debug terminal session that demonstrates breakpoints and viewing variables. User commands are in lower case; system responses are **highlighted**. The program being executed is shown in Figure 43.

```
myprog debug/


 Debug(MYPROG <MAIN-PROGRAM> ):

break 14

 Debug(MYPROG <MAIN-PROGRAM> ):

go


 STOPPED AT MYPROG/<MAIN-PROGRAM>/14
```

Figure 44. Starting a program and setting a breakpoint

```
 Debug(MYPROG <MAIN-PROGRAM> ):
,c
  C = 'A'


 Debug(MYPROG <MAIN-PROGRAM> ):
,hue
  HUE = GREEN


 Debug(MYPROG <MAIN-PROGRAM> ):
,arr[arr[1,1,1].int,1,1].int
  ARR[ARR[1,1,1].INT,1,1].INT = 5


 Debug(MYPROG <MAIN-PROGRAM> ):
,arr[1]
  ARR[1]
 (00020410)
 000000 00000003 00000004 00000004 00000005 '................'
 000010 00000005 00000006 00000006 00000007 '................'


 Debug(MYPROG <MAIN-PROGRAM> ):
,spac[4]
  SPAC[4] = 4


 Debug(MYPROG <MAIN-PROGRAM> ):
,rp->.x
  RP->.X = 4.5


 Debug(MYPROG <MAIN-PROGRAM> ):
,rp->.b
  RP->.B = 3
```

Figure 45. Viewing some program variables

```
 Debug(MYPROG <MAIN-PROGRAM>):
,c (attr
 VARIABLE TYPE: CHAR
 MEMORY CLASS : LOCAL AUTO
 DECLARED IN : <MAIN-PROGRAM>
    C = 'A'


 Debug(MYPROG <MAIN-PROGRAM>):
,arr[1,1,1].int (attr
 VARIABLE TYPE: INTEGER
 MEMORY CLASS : STATIC
 DECLARED IN : <MAIN-PROGRAM>
  ARR[1,1,1].INT = 3


 Debug(MYPROG <MAIN-PROGRAM>):
,spac (attr
 VARIABLE TYPE: ARRAY
 LENGTH    : 40
 MEMORY CLASS : EXTERNAL
 DECLARED IN : <MAIN-PROGRAM>
  SPAC
 (000382F0)
 000000 00000000 00000004 00000008 0000000C '................'
 000010 00000010 00000014 00000018 0000001C '................'
 000020 00000020 00000024                   '........        '


 Debug(MYPROG <MAIN-PROGRAM>):
,rp (attr
 VARIABLE TYPE: POINTER
 MEMORY CLASS : LOCAL AUTO
 DECLARED IN : <MAIN-PROGRAM>
  RP = 000486F8


 Debug(MYPROG <MAIN-PROGRAM>):
,rp-> (attr
 VARIABLE TYPE: RECORD
 LENGTH    : 36
 MEMORY CLASS : DYNAMIC
 DECLARED IN : <MAIN-PROGRAM>
  RP->
 (000486F8)
 000000 07D5C5E6 40D9C5C3 00000000 00000000 '.NEW REC........'
 000010 00000003 00000000 41480000 00000000 '................'
 000020 00000000                            '....            '


 Debug(MYPROG <MAIN-PROGRAM>):
,rp->.a (attr
 VARIABLE TYPE: STRING
 LENGTH    : 7
 MEMORY CLASS : DYNAMIC
 DECLARED IN : <MAIN-PROGRAM>
  RP->.A = 'NEW REC'
```

Figure 46. Viewing variables using the ATTR option

```
 Debug(MYPROG <MAIN-PROGRAM>):
,rp->.junk
 ,RP->.JUNK
         $
 JUNK IS NOT A RECORD FIELD


 Debug(MYPROG <MAIN-PROGRAM>):
,c->
 ,C->
   $
 -> FOLLOWED NON POINTER


 Debug(MYPROG <MAIN-PROGRAM>):
,arr[1,10000,1]
 ,ARR[1,10000,1]
            $
 ARRAY INDEX OUT OF BOUNDS


 Debug(MYPROG <MAIN-PROGRAM>):
go
END OF PROGRAM
```

Figure 47. Debug error messages

This section describes the rules that the Pascal/VS compiler employs in mapping variables to storage locations.

10.1 AUTOMATIC STORAGE

Variables declared locally to a routine via the **var** construct are assigned offsets within the routine's dynamic storage area (DSA). There is a DSA associated with every routine of the program plus one for the main program itself. The DSA of a routine is allocated when the routine is called and is deallocated when the routine returns.

10.2 INTERNAL STATIC STORAGE

For source modules that contain variables declared STATIC, a single unnamed control section ('private code') is associated with the source module in the resulting text deck. Each variable declared via the STATIC construct, regardless of its scope, is assigned a unique offset within this control section.

10.3 DEF STORAGE

Each **def** variable which is initialized by means of the **value** declaration will generate a named control section (csect). Each **def** variable which is not initialized will generate a named common section. The name of the section is derived from the first eight characters of the variable's name.

10.4 DYNAMIC STORAGE

Pointer qualified variables are allocated dynamically from heap storage by the procedure 'NEW'. Such variables are always aligned on a doubleword boundary.

10.5 RECORD FIELDS

Fields of records are assigned consecutive offsets within the record in a sequential manner, padding where necessary for boundary alignment. Fields within unpacked records are aligned in the same way as variables are aligned. The fields of a packed record are aligned on a byte boundary regardless of their declared type.

10.6 DATA SIZE AND BOUNDARY ALIGNMENT

A variable defined in an Pascal/VS source module is assigned storage and aligned according to its declared type.

10.6.1 The Predefined Types

The table in Figure 48 displays the storage occupancy and boundary alignment of variables declared with a predefined type.

STORAGE MAPPING OF DATA		
DATA TYPE	SIZE in bytes	BOUNDARY ALIGNMENT
ALFA	8	BYTE
ALPHA	16	BYTE
BOOLEAN	1	BYTE
CHAR	1	BYTE
INTEGER	4	FULL WORD
REAL	8	DOUBLE WORD
STRING(len)	len+1	BYTE

Figure 48. Storage mapping for predefined types

10.6.2 Enumerated Scalar

An enumerated scalar variable with 256 or fewer possible distinct values will occupy one byte and will be aligned on a byte boundary. If the scalar defines more than 256 values then it will occupy a half word and will be aligned on a half word boundary.

10.6.3 Subrange Scalar

A subrange scalar that is not specified as packed will be mapped exactly the same way as the scalar type from which it is based.

A packed subrange scalar is mapped as indicated in the table of Figure 49. Given a type definition T as:

```
type
  T = packed i..j;

     and

const
  I = ORD(i);
  J = ORD(j);
```

Range of I .. J	SIZE in bytes	ALIGNMENT
0..255	1	BYTE
-128..127	1	BYTE
-32768..32767	2	HALF WORD
0..65535	2	HALF WORD
otherwise	4	FULL WORD

Figure 49. Storage mapping of subrange scalars

Each entry in the first column in the above table is meant to include all possible sub-ranges within the specified range. For example, the range 100..250 would be mapped in the same way as the range 0..255.

10.6.4 RECORDs

An unpacked record is aligned on a boundary in such a way that every field of the record is properly aligned on its required boundary. That is, records are aligned on the boundary required by the field with the largest boundary requirement. For example, record A below will be aligned on a full word because its field A1 requires a full word alignment; record B will be aligned on a double word because it has a field of type REAL; record C will be aligned on a byte.

```
type
  A= record (*full word aligned*)
         A1 : INTEGER;
         A2 : CHAR
     end;

  B= record (*double word aligned*)
         B1 : A;
         B2 : REAL;
         B3 : BOOLEAN
     end;

  C= record (*byte aligned*)
         C1 : packed 0..255;
         C2 : ALPHA
     end;
```

Figure 50. Alignment of records

Packed records are always aligned on a byte boundary;

10.6.5 ARRAYs

Consider the following type definition:

```
type
  A = array [ s ] of t

where type s is a simple scalar
  and t is any type.
```

A variable declared with this type definition would be aligned on the boundary required for data type 't'. With the exception noted below, the amount of storage occupied by this variable is computed by the following expression:

```
(ORD(HIGHEST(s))-ORD(LOWEST(s))+1)
    * SIZEOF(t)
```

The above expression is not necessarily applicable if 't' represents an unpacked record type. In this case, padding will be added, if necessary, between each element so that each element will be aligned on a boundary which meets the requirements of the record type.

Packed arrays are mapped exactly as unpacked arrays, except padding is never inserted between elements.

A multi-dimensional array is mapped as an array of array(s). For example the following two array definitions would be mapped identically in storage.

```
array [ i..j, m..n ] of t

array [ i..j ] of
array [ m..n ] of t
```

10.6.6 FILEs

File variables occupy 64 bytes and are aligned on a full word boundary.

10.6.7 SETs

SETs are represented internally as a string of bits: one bit position for each value that can be contained within the set.

To adequately explain how sets are mapped, two terms will need to be defined: The base type is the type to which all members of the set must belong. The fundamental base type represents the non-subrange scalar type which is compatible with all valid members of the set. For example, a set which is declared as

```
set of '0'..'9'
```

has the base type defined by '0'..'9'; and a fundamental base type of CHAR.

Any two unpacked sets which have the same fundamental base type will be mapped identically (that is, occupy the same amount of storage and be aligned on the same boundary). In other words, given a set definition:

```
type
  S = set of s;
  T = set of t;
```

where s is a non-subrange scalar type and t is a subrange of s: both S and T will have the same length and will be aligned in the same manner.

Sets always have zero origin; that is, the first bit of any set corresponds to a member with an ordinal value of zero (even though this value may not be a valid set member).

Unpacked sets will contain the minimum number of bytes necessary to contain the largest value of the fundamental base type. Packed sets occupy the minimum number of bytes to contain the largest valid value of the base type. Thus, variables A and B below will both occupy 256 bits.

```
var
  A : set of CHAR;
  B : set of '0'..'9';
```

Variables C and D will both occupy 16 bits; variable E will occupy 8 bits.

```
var
  C : set of (C1,C2,C3,C4,C5,C6,
              C7,C8,C9,C10,C11,C12
              C12,C13,C14,C15,C16);
  D : set of C1..C8;
  E : packed set of C1..C8;
```

A set type with a fundamental base type of INTEGER is restricted so that the largest member to be contained in the set may not exceed the value 255; therefore, such a set will occupy 256 bits.

Thus, variables U and V below will both occupy 256 bits; variable W will occupy 21 bits; variable X will occupy 32 bits.

```
var
  U : set of 0..255;
  V : set of 10..20;
  W : packed set of 10..20;
  X : packed set of 0..31;
```

Given that M is the number of bits required for a particular set, the table in Figure 51 indicates how the set will be mapped in storage.

Range of M	SIZE BYTES	ALIGNMENT
1 <= M <= 8	1	BYTE
9 <= M <= 16	2	HALF WORD
17 <= M <= 24	3	FULL WORD
25 <= M <= 32	4	FULL WORD
33 <= M <= 256	(M+7) DIV 8	BYTE

Figure 51. Storage mapping of SETS

10.6.8 SPACEs

A variable declared as a **space** is aligned on a byte boundary and occupies the number of bytes indicated in the length specifier of the type definition. For example, the variable S declared below occupies 1000 bytes of storage.

```
var S: space [1000] of INTEGER;
```

11.0 CODE GENERATION FOR THE IBM/370

11.1 LINKAGE CONVENTIONS

Pascal/VS uses standard OS linkage conventions with several additional restrictions. The result is that Pascal/VS may call any program that requires standard conventions and may be called by any program that adheres to the additional Pascal/VS restrictions.

On entry to a Pascal/VS routine the contents of relevant registers are as follows:

- Register 1 - points to the parameter list
- Register 12 - points to the Pascal/VS Communication Work Area (PCWA)
- Register 13 - points to the save area provided by the caller
- Register 14 - return address
- Register 15 - entry point of called routine

Pascal/VS requires that the parameter register (R1) be pointing into the Dynamic Storage Area (DSA) stack in such a way that 144 bytes prior to the R1 address is an available save area.

11.2 REGISTER USAGE

The table in Figure 52 describes how each general register is used within a Pascal/VS program. The floating point registers are used for computation on data of type REAL.

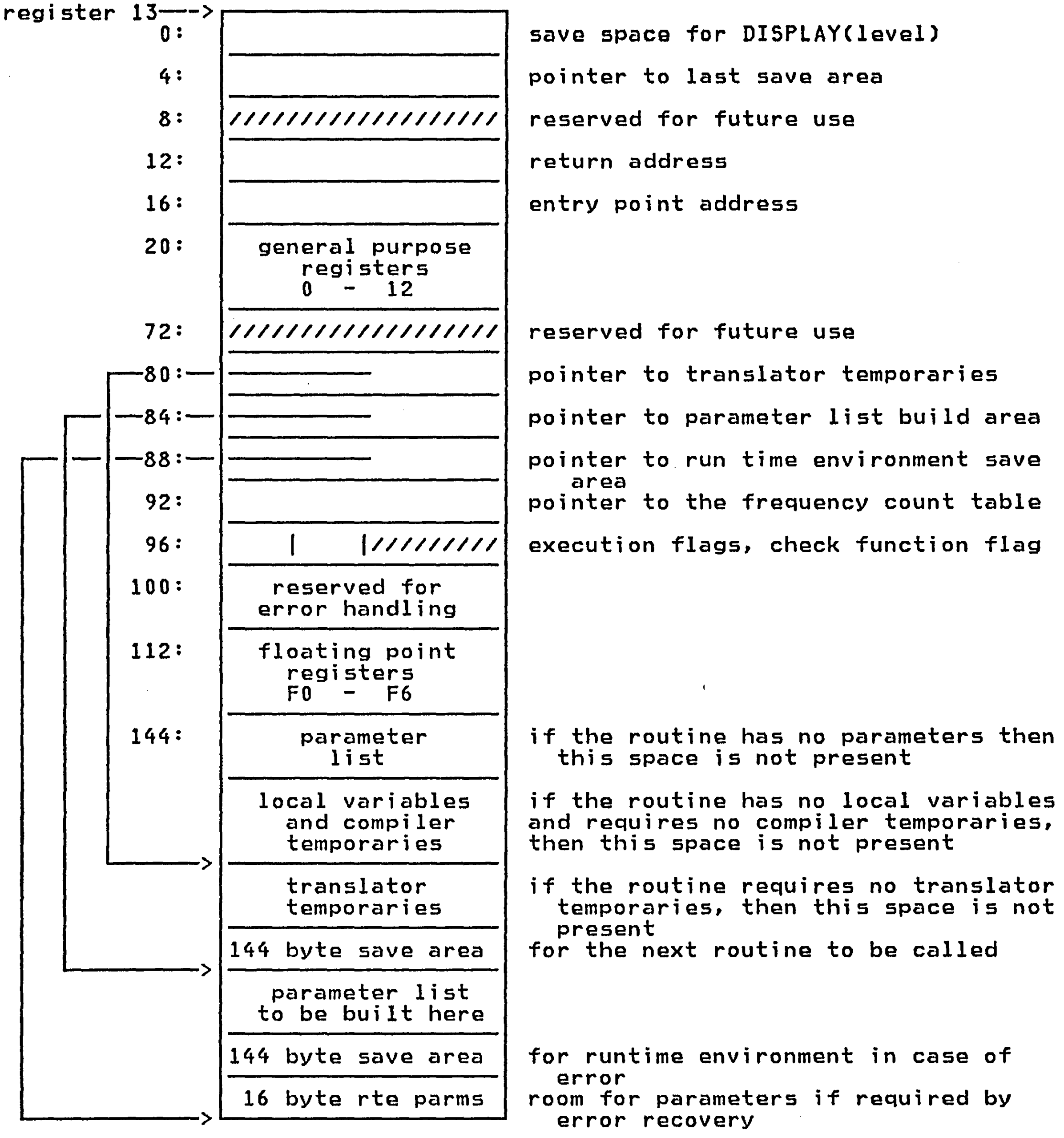
register(s)	purpose(s)
0,1	- temporary work registers for the compiler - standard linkage usage on calls
3,4,5,6,7,8,9	- registers assigned by the compiler for computation and for data base registers
2,10	- code base registers of the currently executing routine
11	- address of DSA of active routine at outermost lexical level
12	- always points to Pascal/VS Communication Work Area
13	- always points to the local DSA
14,15	- temporary work registers for the compiler - standard linkage usage on calls

Figure 52. Register usage

11.3 DYNAMIC STORAGE AREA

On entry to a procedure or function, an area of memory called a Dynamic Storage Area (DSA) is allocated. This area is used to contain save areas, local variables and compiler generated temporaries. Pascal/VS requires a minimum DSA of 144 bytes; if the routine has parameters or local variables, more space is needed.

The first 72 bytes are generally used according to standard OS linkage conventions. The first word is used to copy the previous data base register at the current procedure nesting level.



register 13--->

Offset	Field	Description
0:		save space for DISPLAY(level)
4:		pointer to last save area
8:	////////////////////	reserved for future use
12:		return address
16:		entry point address
20:	general purpose registers 0 - 12	
72:	////////////////////	reserved for future use
80:		pointer to translator temporaries
84:		pointer to parameter list build area
88:		pointer to run time environment save area
92:		pointer to the frequency count table
96:	| |//////////	execution flags, check function flag
100:	reserved for error handling	
112:	floating point registers F0 - F6	
144:	parameter list	if the routine has no parameters then this space is not present
	local variables and compiler temporaries	if the routine has no local variables and requires no compiler temporaries, then this space is not present
	translator temporaries	if the routine requires no translator temporaries, then this space is not present
	144 byte save area	for the next routine to be called
	parameter list to be built here	
	144 byte save area	for runtime environment in case of error
	16 byte rte parms	room for parameters if required by error recovery

//// = indicates that the field is not presently used.

Figure 53. DSA format

11.4 ROUTINE INVOCATION

Each invocation of a Pascal/VS routine must acquire a dynamic storage area (DSA) (see "Dynamic Storage Area" on page 74). This storage is allocated and deallocated in a LIFO (last in/first out) stack. If the stack should become filled to its capacity, a storage overflow routine will attempt to obtain another stack from which storage is to be allocated.

Every DSA must be at least 144 bytes long; this is the storage required by Pascal/VS for a save area. The routine's local variables and parameters are mapped within the DSA starting at offset 144.

Upon entering a routine, register 1 points 144 bytes into the routine's DSA, which is where the parameters passed in by the caller reside. This implies that the calling routine is responsible for allocating a portion of the DSA required by the routine being called, namely 144 bytes plus enough storage for the parameter list. This portion of storage is actually an extension of the caller's DSA.

In general, the DSA of a routine consists of five sections:

1. The local save area (144 bytes).

2. Parameters passed in by the caller.

3. Local variables required by the routine.

4. A save area required by any routine that will be called.

5. Storage for the largest parameter list to be built for a call.

Sections 1 and 2 are allocated by the calling routine; sections 3, 4, and 5 are allocated by the prologue of the routine to which the DSA belongs.

Upon invocation, register 13 points to the base of the DSA of the caller, which is where the caller's save area is located. The new value of register 13 may be computed by subtracting 144 from the value in register 1. Figure 54 illustrates the condition of the stack and relevant registers immediately at the start of a routine.

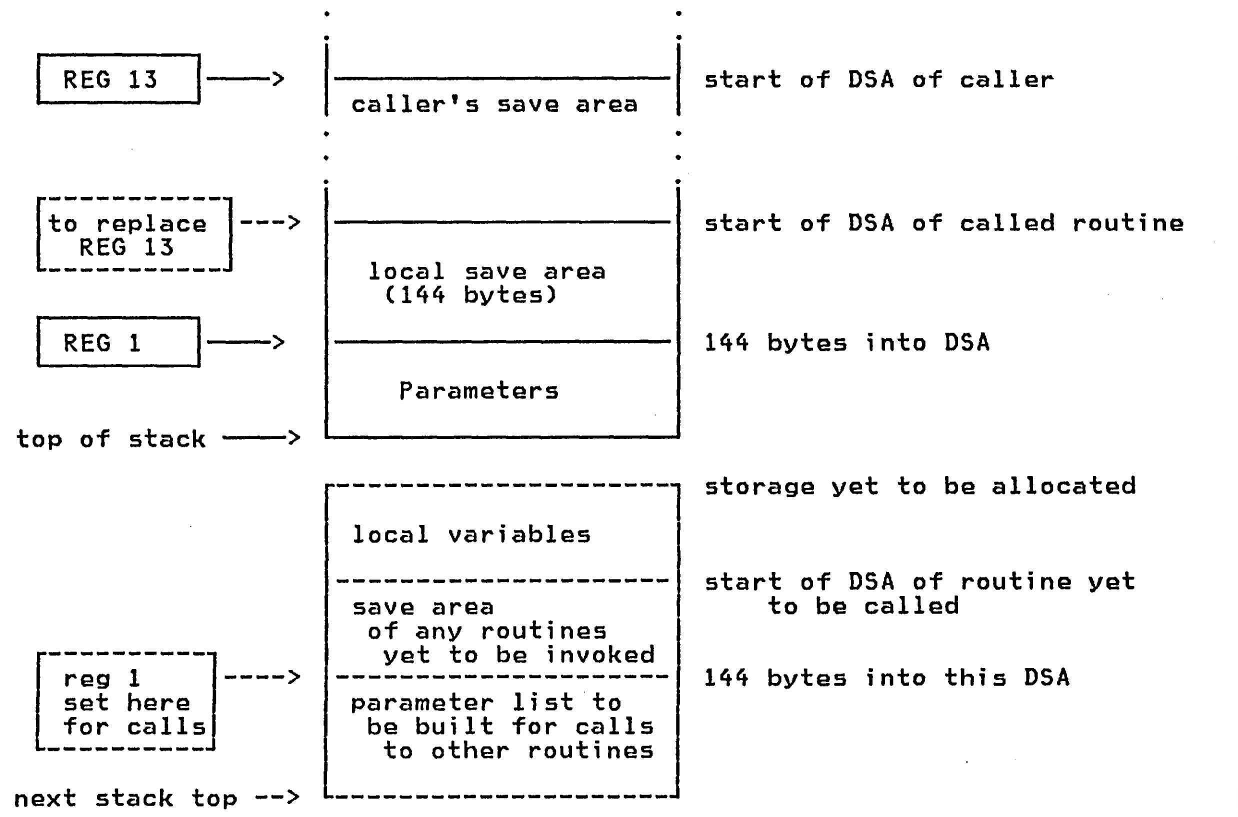

REG 13
caller's save area
start of DSA of caller
to replace REG 13
start of DSA of called routine
local save area (144 bytes)
REG 1
144 bytes into DSA
Parameters
top of stack
storage yet to be allocated
local variables
start of DSA of routine yet to be called
save area of any routines yet to be invoked
reg 1 set here for calls
144 bytes into this DSA
parameter list to be built for calls to other routines
next stack top

Figure 54. Snapshot of stack and relevant registers at start of routine

11.5 PARAMETER PASSING

Pascal/VS passes parameters in several different ways depending on how the parameter was declared. In every case, register 1 contains the address of the parameter list.

The parameter list is aligned on a doubleword boundary and each parameter is aligned on its proper boundary. Addresses are aligned on word boundaries.

11.5.1 Passing by Read/Write Reference

This mechanism is indicated by use of the reserved word **var** in the routine heading. Actual parameters passed in this way may be modified by the invoked routine.

The parameter list contains the address of the actual parameter.

Routine Heading:

```
procedure PROC(var I:INTEGER);
```

Routine Invocation:

```
PROC(J);
```

Parameter list:

```
address of J
```

Figure 55. Passing by Read/Write reference

11.5.2 Passing by Read-Only Reference

This mechanism is indicated by use of the reserved word **const** in the routine heading. Actual parameters passed in this way may not be modified by the invoked routine.

The parameter list contains the address of the actual parameter.

Routine Heading:

```
procedure PROC(const I: INTEGER);
```

Routine Invocation:

```
PROC(J+5);
```

Parameter list:

```
address of a memory location
which contains the value of
J+5.
```

Figure 56. Passing by Read-only reference

11.5.3 Passing by Value

This mechanism is the default way in which parameters are passed. Parameters passed in this way are treated as if they are pre-initialized local variables in the invoked routine. Any modification to these parameters by the invoked routine will not be reflected back to the caller. If the actual parameter is a scalar, pointer, or **set**, then the parameter list will contain the value of the actual parameter. If the actual parameter is an **array**, **record**, **space**, or &string., then the parameter list will contain the address of the actual parameter. In the latter case, the called procedure will copy the parameter into its local storage.

Routine Heading:

```
procedure PROC(
  I : INTEGER;
  A : ALPHA);
```

Routine Invocation:

```
PROC(J,'alpha');
```

Parameter list:

```
value of J
address of 'alpha          '
```

Figure 57. Passing by value

11.5.4 Passing Procedure or Function Parameters

For procedures or functions which are being passed as parameters, the address of the routine is placed in the parameter list.

```
Routine Heading:

  procedure PROC(
    function X(Y: REAL): REAL );

Routine Invocation:

  PROC(COS);

Parameter list:

  address of COS routine
```

Figure 58. Passing routine parameters

11.5.5 Function Results

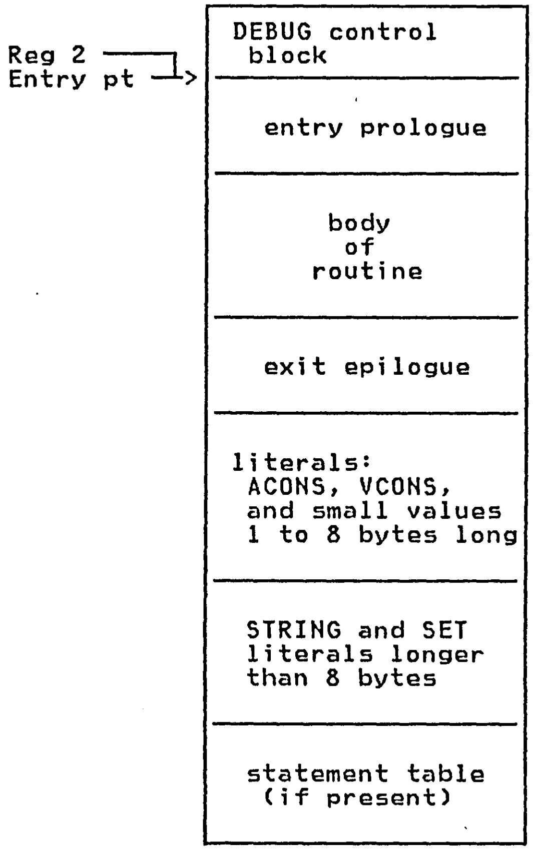
Pascal/VS functions have an implicit parameter which precedes all specified parameters. This parameter contains the address of the memory location where the function result is to be placed.

```
Routine Heading:

  function FUNC(C: CHAR):INTEGER;

Routine Invocation:

  I := FUNC('L');

Parameter list:

- address of returned integer
  result
- value of character 'L'
```

Figure 59. Function results

11.6 PROCEDURE/FUNCTION FORMAT

Every Pascal/VS procedure or function is arranged in the order shown below. Register 2 is the code base register for the first 4K bytes of the routine body. If the routine occupies more than 4K bytes, register 10 is used as the code base register for the second 4K bytes. If a routine exceeds 8K bytes of storage, the compiler will diagnose it as a terminal error.



Reg 2
Entry pt
DEBUG control block
entry prologue
body of routine
exit epilogue
literals: ACONS, VCONS, and small values 1 to 8 bytes long
STRING and SET literals longer than 8 bytes
statement table (if present)

Figure 60. Routine format

11.7 PCWA

The Pascal Communications Work Area is always addressable from register 12. This area of memory is used to contain global information about the execution of the program.

The area is divided into two parts, each is 2048 bytes in length. The first part contains data that needs to be addressable; the second is composed of the small routines used to augment the generated code. An example is the routine that is used to concatenate two strings.

offset		width in bytes
0	end of stack	4
4	current stack	4
8	flags 1	4
12	flags 2	4
16	return code	4
20	pointer to files	4
24	pointer to parms	4
28	module link	4
32	ext. save area	4
36	level display	32
68	debug temp	4
72	floating pt temp	8
80	conversion const1	8
88	conversion const2	8
96	set mask	8
104	temp dsa save	8
112	error recovery save area	144
256	error recovery param list build	64
320	address of HALT	4
324	addr of allocator	4
328	addr of dealloc	4
332	default alloc size	4
336	addr of checker	4
340	reserved	1436
1776	SPIE save area	144
1920	SPIE work area	64
1984	memory space desc	64

Figure 61. Pascal Communications Work Area

end of stack
a pointer to the end of the current DSA stack.

current stack
a pointer to the top of the current DSA stack.

flags 1
reserved for future use.

flags 2
flags used to enable runtime features.

return code
the value assigned by the last execution of RETCODE or zero if RETCODE has not been called.

pointer to files
a pointer to the first file that has been opened but never closed.

pointer to parms
a pointer to the parameter list passed to the program.

module link
a pointer to the head of a chain that links modules together as directed by the interactive debugger.

ext. save area
contains the pointer to the save area for the caller of the Pascal program.

level display
a stack of 8 base registers that contain the addresses of the DSAs that are available to the executing routine.

debug temp
a temporary used by the symbolic debugger.

floating pt temp
a temporary used in conversion between floating point numbers and integers.

conversion const1
a constant that contains the floating point value zero.

conversion const2
a constant that contains the floating point value of 2 raised to the 31 power minus 1 in an unnormalized form.

set mask
eight bytes that contain masks used in set operations.

temp dsa save
a temporary used during execution errors.

error recovery save area
used as a register save area when a program error or checking error occurs.

error recovery parm list build
used when a program error or checking error occurs to build a parameter list in order to invoke a recovery procedure.

address of HALT
address of a procedure which terminates the program no matter what state it is in. This procedure is normally HALT.

addr of allocator
address of the routine which is responsible for allocating blocks of storage.

addr of deallocator
address of the routine which releases blocks of storage.

default alloc size
the number of bytes of storage that the allocation routine will allocate when called.

addr of checker
the address of the routine which is invoked to diagnose a checking error.

reserved

spie save area
a small save area used when a SPIE exit is invoked.

spie work area
a place to save certain information from the SPIE.

memory space desc
descriptors used to control the allocation and deallocation policies of dynamic storage and I/O buffers.

11.8 FCB - FILE CONTROL BLOCK

Every Pascal/VS file is represented by a file control block. An FCB is composed of 64 bytes of space.

offset		width in bytes
0	file pointer	4
4	flags	2
6	elem len	2
8	symbolic name	8
16	buf idx	2
18	buf end	2
20	rec len	2
22	rec end	2
24	pointer to buffer	4
28	pointer to record	4
32	last fcb	4
36	next fcb	4
40	pointer to DCB	4
44	pointer to DECB	4
48	aux buffer	4
52	pointer to exten.	4
56	current status	4
60	not assigned	4

Figure 62. File Control Block (FCB) format

The fields are defined as:

File pointer
points to the current element of the file.

Flags
set of file flags (16 bits). The flags are:

FOPEN indicates that file is open;

FINPUT the file is open for input (output otherwise);

FTEXT the file is of type TEXT;

FEOLN end-of-line condition is true;

FEOF	end-of-file condition is true;
FFIXED	file is fixed block (variable block otherwise);
FSEQ	sequential file;
FINTER	interactive file;
FFEOL	end-of-line condition is true, but not as a result of READLN;
FSUMR	file is prepared for reading;
FSUMW	file is prepared for writing;
FALTIO	alternate I/O system in use.

Elem len
the length of one element of the file

Symbolic name
the DDNAME of the file.

Buf idx
count of the number of bytes from beginning of buffer used.

Buf end
total length of buffer in bytes.

Rec len
logical record length of current record.

Rec end
byte offset from beginning of buffer for the end of the current record.

Pointer to buffer
address of the beginning of the buffer.

Pointer to record
address of the current record in the buffer.

Last FCB
back chain of currently open FCBs.

Next FCB
forward chain of currently open FCBs.

Pointer to DCB
address of the OS Data Control Block.

Pointer to DECB
address of the Data Event Control Block.

Aux buffer
the address of a buffer that needs to be freed when the file is closed.

Pointer to exten.
the address of another 64 byte area used to implement special IO interfaces.

Current status
status of the file.

12.0 LINKING TO ASSEMBLER ROUTINES

Writing an assembler language routine for Pascal/VS is a simple operation provided that a set of conventions are carefully followed. There are two reasons for the need for these conventions:

1. Pascal/VS parameter passing conventions: As described in "Parameter Passing" on page 76, Pascal/VS parameters are passed in a variety of ways, depending on their attributes.

2. The Pascal/VS environment: This is an arrangement of registers and control blocks used by Pascal/VS to handle storage management and runtime error recovery. (see "Register Usage" on page 73.)

12.1 WRITING ASSEMBLER ROUTINE WITH MINIMUM INTERFACE

Writing an assembler routine with the minimum interface requires the least knowledge of the runtime environment. However, such a routine has the following deficiencies:

- It may not call a Pascal/VS routine;

- It must be non-recursive;

- If a program error should occur (such as divide by zero), the Pascal/VS runtime environment will not recover properly and the results will be unpredictable.

When a Pascal/VS program invokes an assembler language routine, register 14 contains the return address and register 15 contains the starting address of the routine. The routine must follow the System/370 linkage conventions and save the registers that will be modified in the routine. It must also save any floating point register that is altered in the routine.

Upon entry to the routine, register 13 will contain the address of the register save area provided by the caller, and register 1 will point to the first of a list of parameters being passed (if such a list exists). Once the register values are stored in the caller's save area, the save area address (register 13) must be stored in the backchain word in a save area defined by the assembler routine itself. Before returning to the Pascal/VS routine, the registers must be restored to the values that they contained when the assembler routine was invoked.

If you insert your assembler instructions at the point indicated in the skeletal code shown in Figure 63, your assembler routine can be called from a Pascal/VS routine and you need have no knowledge of the Pascal/VS environment.

```
anyname  CSECT
         ENTRY procname      declare routine name as an entry point
procname DS    0H              entry point to routine
         STM   14,12,12(13)    save Pascal/VS registers in Pascal/VS save area
         BALR  basereg,0       establish base register
         USING *,basereg
         ST    13,SAVEAREA+4   store Pascal/VS save area address
         LA    13,SAVEAREA     load address of local save area
         .
         .                     body of assembler routine
         .
*                              restore the floating point registers if
*                              they were saved
         L     13,4(13)        restore Pascal/VS registers
         LM    14,12,12(13)
         BR    14              return to Pascal/VS
SAVEAREA DC    20F'0'          local save area
         END
```

Figure 63. Minimum interface to an assembler routine: skeletal code to be invoked from Pascal/VS

12.2 WRITING ASSEMBLER ROUTINE WITH GENERAL INTERFACE

```
procname PROLOG  LASTREG=r,VARS=n,PARMS=p

         EPILOG  LASTREG=r
```

where:

procname is the entry point name of the routine.

LASTREF is a number between 3 and 12, inclusive, which indicates the highest register to be modified by the routine between 3 and 12. This value must be the same for both the PROLOG and EPILOG macros.

VARS is the number of bytes required for any local data, including passed-in parameters.

PARMS is the number of bytes required for the largest parameter list to be built within the routine.

defaults:
LASTREG=12
VARS=3
PARMS=0

Figure 64. PROLOG/EPILOG macros

If an assembler routine has at least one of the following characteristics, the general interface must be used:

- It calls a Pascal/VS routine;
- It is recursive;
- Program errors must be intercepted and diagnosed by the Pascal/VS runtime environment.

Two assembler macros are available which are used to generate the prologue and epilogue of an assembler routine with a general Pascal/VS interface. The macro names are PROLOG and EPILOG and their forms are described in the figure above.

The PROLOG macro preserves any registers that are to be modified and allocates storage for the DSA. It also includes code to recover from a stack overflow and program error. The label of the macro is established as an ENTRY point; register 2 is established as the base register for the first 4 kilobytes of code.

Upon entering a routine prior to executing the PROLOG code, the following registers are expected to contain the indicated data:

- Register 1 - address of the parameter list built by the caller, which is 144 bytes into the DSA to be used by the called routine.
- Register 12 - address of the Pascal Communication Work Area (PCWA).
- Register 13 - address of the DSA of the calling routine.
- Register 14 - return address.
- Register 15 - address of the start of the called routine.

Upon executing the code generated by the PROLOG macro, the registers are as follows:

- Register 0 - unchanged
- Register 1 - address of an area of storage in which parameter lists may be built to pass to other routines.
- Register 2 - base register for the first 4 kilobytes of code within the invoked routine.
- Registers 3 through 11 - unchanged.
- Register 12 - unchanged
- Register 13 - address of the local DSA of the routine just invoked. The first 144 bytes is the register save area for the invoked routine. Following the save area is where the parameters passed in by the caller are located. Immediately after the parameters is storage for local variables followed by a parameter list build area.
- Register 14 - unchanged.
- Register 15 - unpredictable.

The EPILOG macro restores the saved registers, then branches back to the calling routine. In order for the epilogue to execute properly, register 13 must have the same contents as was established by the prologue.

The contents of the floating point registers are not saved by the PROLOG macro. If the floating point registers are modified, they must be restored to their original contents prior to returning from the routine.

A skeleton of a general-interface assembler language routine which may be called by a Pascal/VS program is given below.

```
* The following names have the indicated meaning
* 'csectnam' is the name of the csect in which the routine resides
* 'procname' is the name of the routine.
* 'parmsize' is the length of the passed-in parameters
* 'varsize' is the storage required for the local variables
* 'lastreg' is the highest register (up to 12) which will be modified
* 'plist' is the length of the largest parameter list required for calls
*         to other routines from "procname"
*
csectnam CSECT
*
procname PROLOG LASTREG=lastreg,VARS=varsize+parmsize,PARMS=plist
         .
         .                      <== insert code here
         .
*
         EPILOG LASTREG=lastreg
         END
```

Figure 65. General interface to an assembler routine: skeletal code to be invoked from Pascal/VS

12.3 RECEIVING PARAMETERS FROM ROUTINES

Parameters received from a Pascal/VS routine are mapped within a list in the manner described in "Parameter Passing" on page 76. At invocation register 1 contains the address of this list.

If the general interface (see "Writing Assembler Routine with General Interface" on page 84) is used in writing the assembler routine, passed-in parameters start at offset 144 from register 13 after the prologue has been executed.

12.4 CALLING PASCAL/VS ROUTINE FROM ASSEMBLER ROUTINE

An assembler language routine may call a Pascal/VS routine provided that:

1. the Pascal/VS runtime environment is active (this will be so if the assembler routine was invoked by a Pascal/VS procedure),
2. the general Pascal/VS interface was incorporated, and
3. the Pascal/VS routine to be called is an ENTRY routine.

Prior to making the call, register 1 must contain the value assigned to it within the PROLOG code. Parameters to be passed are stored into appropriate displacements from register 1 as described in "Parameter Passing" on page 76.

At the point of call, register 12 must contain the address of the Pascal Communications Work Area (PCWA). This will be the case if the assembler routine was invoked from a Pascal/VS routine and has not modified the register.

To perform the call, a V-type constant address of the routine to be called is loaded into register 15 and then the instruction 'BALR 14,15' is executed.

12.5 SAMPLE ASSEMBLER ROUTINE

In Figure 66 on page 87 and Figure 67 on page 87, a sample assembler routine is listed which may be called from a Pascal/VS program. This routine executes an OS TPUT macro to write a line of text to a user's terminal.

```
type
  BUFINDEX = 0..80;
  BUFFER = packed array[1..80] of CHAR;

(*this routine is in assembly language*)

procedure TPUT(
    const BUF : BUFFER;
          LEN : BUFINDEX);
  EXTERNAL;

(*this routine is called from the assembly language routine*)
procedure ERROR(
          RETCODE: INTEGER;
    const MESSAGE: STRING);
  ENTRY;
begin
  WRITELN(OUTPUT, MESSAGE, ', RETURN CODE = ', RETCODE)
end;
```

Figure 66. Pascal/VS description of assembler routine: the assembler routine is shown in Figure 67.

```
TIOSEG   CSECT
TPUT     PROLOG LASTREG=4        only registers 3 and 4 are modified
*
         L      3,144(13)        load address of 'BUF' parameter
         L      4,148(13)        laod value of 'LEN' parameter
         TPUT   (3),(4)          write content of 'BUF' to terminal
         LTR    15,15            check return code
         BZ     TPUTRET          if no error then return
*                                build parm list for call to 'ERROR'
         ST     15,0(1)          assign to 'RETCODE' parameter
         LA     3,TPUTMSG        load address of message
         ST     3,4(1)           assign to 'MESSAGE' parameter
         L      15,=V(ERROR)     load address of 'ERROR' procedure
         BALR   14,15            call 'ERROR'
*
TPUTRET  EPILOG LASTREG=3
*
TPUTMSG  DC     AL1(L'TPUTTEXT)  length byte of string
TPUTTEXT DC     C'TPUT ERROR'    message text
         END
```

Figure 67. Sample assembler routine: this routine is invoked by a Pascal/VS routine and, within itself, invokes a Pascal/VS routine.

12.6 CALLING A PASCAL/VS MAIN PROGRAM FROM ASSEMBLER ROUTINE

A Pascal/VS program may be invoked from an assembler language routine by loading a V-type address constant of the main program name into register 15 and executing a BALR instruction with 14 as the return register.

The convention employed in passing parameters to a program is dependent on whether you are running under CMS or under TSO (or OS Batch). Both conventions require that register 1 be set to the address of the parameter data.

Program to be called:

```
program test;
  ...
begin
  ...
end.
```

Assembler instructions to perform the call under CMS:

```
         ...
         LA   1,PLIST
         L    15,=V(TEST)
         BALR 14,15
         ...
   PLIST DS   0F
         DC   CL8'TEST'
         DC   CL8'token 1'
         DC   CL8'token 2'
            ...
         DC   CL8'token n'
         DC   8X'FF'
```

Assembler instructions to perform the call under VS2 (and TSO):

```
         ...
         LA   1,PLIST
         L    15,=V(TEST)
         BALR 14,15
         ...
   PLIST DS   0F
         DC   XL1'80'       set first bit of address
         DC   AL3(PARMS)
         ...
   PARMS DC   FL2'length'   length of parameter string
         DC   C'parm string goes here'
```

Figure 68. Example of calling a Pascal/VS program from an assembler routine

13.0 RUNTIME ENVIRONMENT OVERVIEW

13.1 PROGRAM INITIALIZATION

Upon invoking a Pascal/VS program, the routine which is responsible for establishing the Pascal/VS execution time environment gains control and performs the following functions:

1. Memory is obtained in which dynamic storage areas (DSA) are allocated and deallocated.
2. The Pascal Communication Work Area (PCWA) is created and initialized.
3. An environment is set up to intercept program interrupts (fixed point overflow, divide by zero, etc.)
4. The main program is called.
5. Upon return from the main program any open files are closed.
6. Acquired memory is freed.
7. Control is returned to the system.

13.2 THE MAIN PROGRAM

The main program is called as an ordinary procedure from the environment setup routine (AMPXSTRT). The external name AMPXBEGN is associated with the address of the main program execution code.

13.3 INPUT/OUTPUT ROUTINES

The I/O operations (which appear as calls to predefined procedures in Pascal/VS) are implemented as calls to internal procedures within the runtime environment.

Internal Input/Output Routines

Procedure name	Action Performed
AMPXRSET	Opens a file
AMPXOPEN	Opens a file by means of OPEN
AMPXCLOS	Closes a file
AMPXRCHR	Reads a character from a text file
AMPXRINT	Reads an integer value from a text file
AMPXRR	Reads a floating point value from a text file
AMPXRSTR	Reads a string from a text file
AMPXRTXT	Reads an array of characters from a text file
AMPXWB	Writes a boolean value to a text file
AMPXWCHR	Writes a character to a text file
AMPXWINT	Writes an integer to a text file
AMPXWR	Writes a real value to a text file
AMPXWSTR	Writes a string to a text file
AMPXWTXT	Writes an array of characters to a text file
AMPXGET	Performs a GET operation on a file
AMPXPUT	Performs a PUT operation on a file
AMPXRREC	Performs a READ operation on a non-text file
AMPXWREC	Performs a WRITE operation on a non-text file

13.4 HEAP MANAGEMENT ROUTINES

The NEW operation generates a call to the internal procedure AMPXNEW. This procedure allocates storage within a heap. If a heap has not yet been created, NEW will obtain memory from the operating system to create a heap.

The DISPOSE operation generates a call to the procedure AMPXDISP. This procedure deallocates the heap storage acquired by a preceding call to AMPXNEW.

The MARK operation generates a call to the procedure AMPXMARK. This procedure creates a new heap from which subsequent calls to AMPXNEW will obtain storage.

The RELEASE operation generates a call to the procedure AMPXRLSE. This procedure frees a heap that was previously created via the AMPXMARK procedure.

Subsequent calls to AMPXNEW will obtain storage from the heap which was active prior to the call of AMPXMARK.

14.0 COMPARISON TO PASCAL

Release 1.0 of Pascal/VS has several differences from 'standard' Pascal. Most of the deviations are in the form of extensions to Pascal in those areas where Pascal does not have suitable facilities.

14.1 PASCAL/VS RESTRICTIONS

Pascal/VS contains the following restrictions that are not in standard Pascal.

Non-local labels
Branching to a non-local label (by means of the **goto** statement) is not supported.

Files
Fields within records and elements of arrays may not be declared as files. Files may not be pointer qualified.

Routine parameters
A routine which is passed as a parameter must not be nested within another routine; that is, it must be at the outermost nesting level.

14.2 MODIFIED FEATURES

Pascal/VS has modified the meaning of a negative length field qualifier on an operand within the WRITE statement.

14.3 NEW FEATURES

Pascal/VS provides a number of extensions to Pascal.

- Separately compilable modules are supported with the **segment** definition.
- 'internal static' data is supported by means of the **static** declarations.
- 'external static' data is supported by means of the **def** and **ref** declarations.
- Static and external data may be initialized at compile time by means of the **value** declaration.
- Constant expressions are permitted wherever a constant is permitted except as the lower bound of a subrange type definition.
- The keyword "**range**" may be prefixed to a subrange type definition to permit the lower value to be a constant expression.
- A varying length character string is provided. It is called STRING.
- The STRING operators and functions are concatenate, LENGTH, STR, SUBSTR, DELETE, TRIM, LTRIM, COMPRESS and INDEX.
- The parameters of the text READ procedure may be length-qualified.
- Calls to FORTRAN subroutines and functions are provided for.
- Input files may be opened as "INTERACTIVE" so that I/O may be done conveniently from a terminal.
- I/O is supported for partitioned data sets.
- Files may be explicitly closed by means of the CLOSE procedure.
- The DDNAME to be associated with a file may be determined at execution time with the OPEN procedure.
- The **space** structure is provided for processing packed data.
- Records may be packed to the byte.
- The tagfield in the variant part of a record may be anywhere within the fixed part of the record.
- Fields of a record may be unnamed.
- Tag specifications on record variants may be ranges (x..y).
- Integers may be declared to occupy bytes and halfwords in addition to full words, as a result of the **packed** qualifier.
- Sets permit the operations of set complement and set exclusive union.
- A function may return any type of data except a **file**.
- The operators '|', '&', '&&' and '¬' may be applied to data of type integer. When applied to integers, the operators act on a bit by bit basis. Shift operations on data are also provided.
- Integer constants may be expressed in hexadecimal digits.

- Real constants (floating point) may be expressed in hexadecimal digits.

- string constants may be expressed in hexadecimal digits.

- The %INCLUDE facility provides a means to include source code from a library.

- A parameter passing mechanism (**const**) has been defined which guarantees that the actual parameter is not modified yet does not require the copy overhead of a pass by value mechanism.

- **leave, continue** and **return** are new statements that permit a branching capability without using a **goto**.

- Labels may be either a numeric value or an identifier.

- **case** statements may have a range notation on the component statements.

- An **otherwise** clause is provided for the **case** statement.

- The variant labels in records may be written with a range notation.

- The **assert** statement permits runtime checks to be compiled into the program.

- The following system interface procedures are supported: HALT, CLOCK, and DATETIME.

- Constants may be of a structured type (namely arrays and records).

- To control the compiler listing, the following listing directives are supported: %PAGE, %SKIP, and %TITLE.

15.0 IMPLEMENTATION SPECIFICATIONS

15.1 SYSTEM DESCRIPTION

The Pascal/VS compiler runs on the IBM System/370 to produce object code for the same system. System/370 includes all models of the 370, 303x, and 43xx computers providing one of the following operating environments:

- VM/CMS
- OS/VS2 TSO
- OS/VS2 Batch

15.2 MEMORY REQUIREMENTS

Under CMS, Pascal/VS requires a virtual machine of at least 768K to compile a program. Execution of a compiled program can be performed in a 256K CMS machine.

The compiler requires a minimum region size of 512K under VS2 (MVS). A compiled and link-edited program can execute in a 128K region.

15.3 IMPLEMENTATION RESTRICTIONS AND DEPENDENCIES

Boolean expressions

Pascal/VS "short circuits" boolean expressions involving the **and** and **or** operators. For example, given that A and B are boolean expressions and X is a boolean variable, the evaluation of

```
X := A or B or C
```

would be performed as

```
if A then
  X := TRUE
else
  if B then
    X := TRUE
  else
    X := C
```

The evaluation of

```
X := A and B and C
```

would be performed as

```
if ¬A then
  X := FALSE
else
  if ¬B then
    X := FALSE
  else
    X := C
```

See the section entitled "Boolean Expressions" in the Pascal/VS Reference Manual for more details.

Floating-point

Some commonly required characteristics of System/370 floating-point arithmetic are shown in Figure 69 on page 94.

Identifiers

Pascal/VS permits identifiers of up to 16 characters in length. If the compiler encounters a longer name, it will ignore that portion of the name longer than 16 characters.

Names of external variables and external routines must be unique within the first 8 characters. Such names may not contain an underscore '_' within the first 8 characters.

Integers

The largest integer that may be represented is 2147483647.[9] This is the value of the predefined constant MAXINT.

The most negative integer that may be represented is -2147483648. This is the value of the predefined constant MININT.

Routine nesting

Routines may be nested up to eight levels deep.

Routines passed as parameters

The following standard routines may not be passed as parameters to another routine:

ABS, CHR, CLOSE, DISPOSE, EOF, EOLN, FLOAT, GET, HBOUND, HIGHEST, INTERACTIVE, LBOUND, LENGTH, LOWEST, MARK, MAX, NEW, ODD, ORD, PACK, PAGE, PRED, PUT, READ, READLN, RELEASE, RESET, REWRITE, ROUND, SIZEOF, SQR, STR, SUCC, TRUNC, UNPACK, WRITE, WRITELN

A routine may not be passed as a parameter if it is nested within another routine; that is, a rou-

[9] This is the highest signed value that may be represented in a 32 bit word.

Floating-point Characteristics

Characteristic	Decimal approximation	Exact Representation[1]
Maxreal[2]	7.23700557733226E+75	'7FFFFFFFFFFFFFFF'XR
Minreal[3]	5.39760534693403E-79	'0010000000000000'XR
Epsilon[4]	1.38777878078145E-17	'3310000000000000'XR

[1] The syntax '...'XR is the way hexadecimal floating-point numbers are represented in Pascal/VS. See the section entitled "Constants" in the Pascal/VS Reference Manual.

[2] Maxreal is the largest finite floating-point number that may be represented.

[3] Minreal is the smallest positive finite floating-point number that may be represented.

[4] Epsilon is the smallest positive floating-point number such that the following condition holds:

```
1.0+epsilon > 1.0
```

This value is often needed in numerical computations involving converging series.

Figure 69. Characteristics of System/370 floating point arithmetic

tine being passed as a parameter must be at the outermost nesting level.

A FORTRAN function or subroutine may not be passed as a parameter to a Pascal/VS routine.

Sets

Given a **set** type of the form

```
set of a..b
```

where "a" and "b" express the lower and upper bounds of the base scalar type, the following conditions must hold:

- ORD(a) >= 0
- ORD(b) <= 255

16.1 PASCAL/VS COMPILER MESSAGES

No.	Message and Explanation
0	**Not yet implemented** The indicated construct is not currently implemented.
1	**Identifier expected**
2	**Source continues after end of program** The compiler detected text after the logical end of the program. This error is often caused by mismatched **begin/end** brackets.
3	**"END" expected**
4	**Character in quoted string is not displayable** The indicated character within a quoted string does not correspond to a valid displayable EBCDIC character. If the string is printed on a device, the character may be interpreted as a control character that could cause unpredictable results. If a control character is intended, then the string should be represented in hexadecimal form.
5	**Symbol invalid or out of context** The indicated symbol is not part of the syntax of the construct being scanned. The symbol should be deleted or changed.
6	**EOF before logical end of program** The compiler came to the end of the source program before the logical end of the program was detected. This error is often caused by mismatched **begin/end** brackets.
7	**"BEGIN" expected**
8	**semicolon ';' expected**
9	**VAR declarations not permitted here** The indicated **var** declaration appears in the outermost lexical level of a segment module. Automatic variables (those declared via the **var** construct) must be local to either the main program or to a routine; they may not be declared in the outermost level of a segment module. The declaration may be changed to **static**.
11	**Ambiguous procedure/function specification** The routine directive EXTERNAL or FORTRAN was applied to the indicated routine declaration that was also declared as an ENTRY routine. Such a combination is contradictory.
12	**Multiply defined label** The indicated label has been previously defined within the surrounding routine.

13	**Label identifier expected** Within the indicated label definition, a label identifier is missing. A label identifier is either an alphanumeric identifier or an integer constant within the range 0 to 9999.
14	**File types restricted to simple variables** Only a variable may be declared as a file. As a restriction imposed by Pascal/VS, neither a field of a record nor the elements of an array may be declared as a file. In addition, the object of a pointer may not be of a file type.
15	**'=' expected**
16	**Identifier required to be a type in tag field specification** Within a **record** definition, a tag field is being declared, but the indicated identifier which is supposed to represent the tag field's type was not declared as a type.
17	**':' expected**
18	**Parameters on forwarded routine not necessary** A routine declaration which has been previously declared as FORWARD must not specify any formal parameters. Any formal parameters are assumed to have been specified previously on the associated declaration that contained the FORWARD directive.
19	**Files passed by value not permitted** The indicated formal value parameter is of a file type. A file variable may be passed to a routine only by the **var** or **const** mechanism; never by value.
21	**')' expected**
22	**Forwarded routine class conflict** A procedure declaration was previously declared as a forwarded function; or a function declaration was previously declared as a forwarded procedure.
23	**Routine nesting exceeds maximum** The indicated **procedure** or **function** declaration exceeds the maximum allowed nesting level for routines. Routines may be nested to a maximum depth of 8.
24	**Too many nested WITH statements or RECORD definitions** This error is caused by either too many nested **with** statements, or too many nested **record** definitions.
25	**Type not needed on forwarded function** A function declaration which has been previously FORWARDed must not specify a return type. The type specification is assumed to have been specified previously on the associated declaration that contained the FORWARD directive.
26	**Missing type specification for function** The indicated function header did not specify a return type.

27	**Procedure/Function previously FORWARDed** The indicated routine declaration that contains the FORWARD directive was already previously forwarded.
28	**Additional errors not printed** The indicated construct contained more errors, but were not printed due to space considerations.
29	**Illegal hexadecimal or binary digit** An invalid hexadecimal digit was detected within a hexadecimal constant specification of the form '...'X, '...'XC, or '...'XR; or, an invalid binary digit was detected within a binary constant specification of the form '...'B. The following characters are valid hexadecimal digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, a, b, c, d, e, f The following characters are valid binary digits: 0, 1
30	**Unidentifiable character** The indicated character is not recognized as a valid token.
31	**Digit expected** A decimal digit was expected but missing at the indicated location.
32	**Real constant has too many digits** The indicated floating point constant contains more digits than the compiler allows for in scanning. If this error should occur, please notify the compiler maintenance group at IBM.
33	**Integer constant too large** The indicated integer constant is not within the range -2147483648 to 2147483647.
34	**End of string not seen** A string constant may not cross a line boundary. This error is often the result of mismatched quotes. If a string constant is too large to fit on one line, it must be broken up into multiple strings and concatenated with the || operator. (Concatenation of string constants is performed at compile time).
35	**Hexadecimal integer constant may not exceed 8 digits** The indicated hexadecimal constant exceeds the maximum allowed number of digits.
36	**Char string is too large** The indicated string constant exceeds 255 characters, which is the implementation limit. This may happen when multiple string constants are concatenated.

37	**Standard routines not permitted as parameters** Standard routines which generate in line code may not be passed as parameters to other routines. The following is a list of such routines: ABS, CHR, CLOSE, DISPOSE, EOF, EOLN, FLOAT, GET, HBOUND, HIGHEST, INTERACTIVE, LBOUND, LENGTH, LOWEST, MARK, MAX, NEW, ODD, ORD, PACK, PAGE, PRED, PUT, READ, READLN, RELEASE, RESET, REWRITE, ROUND, SIZEOF, SQR, STR, SUCC, TRUNC, UNPACK, WRITE, WRITELN
38	**Variable must be of type file** The indicated variable is required to be of a file type.
39	**Must be of type TEXT** The indicated variable is required to have been declared with the predefined type TEXT.
40	**Required parameters are missing** The indicated READ or WRITE statement contains no parameter from which to reference data.
41	**Comma ',' expected**
42	**User defined scalars not permitted** Expressions which are of a user defined enumerated type may not be directly read from or written to a text file.
43	**Operand of READ/WRITE not of a valid type** Any parameter passed to the procedures READ or WRITE (text file case) must be compatible with one of the following types: - INTEGER - REAL - CHAR - BOOLEAN - STRING - **packed array**[1..n] **of** CHAR where n is a positive integer constant.
44	**Length field must be integer** The indicated length qualifier expression in a READ or WRITE statement is not of type integer. Any length specification within a text-file READ/WRITE must be of type integer.

45	**Set contains constant member(s) which are out of range**
	The indicated set constant contains members which are not valid for the set variable to which the constant is being assigned.
	For example,
	`var S : set of 10..20;`
	`begin`
	`  S := [1,2]; (*<== this statement would produce error 45*)`
	`end;`
	This error may also occur when a set constant is being passed as a parameter.

46	**2nd length applicable only to REAL data** In the procedure WRITE (text file case), only expressions of type REAL are permitted to have two length field qualifications.

48	**Associated variable of subscript must be of an array type** An attempt is being made to subscript a variable which was not declared as an array.
49	**Expression must be of a simple scalar type** The indicated expression should be of a simple scalar type within the context in which it is being used.
51	**Variable must be of a pointer type** The indicated variable is being used as a pointer; however, the variable was not declared as being of a pointer type.
52	**Corresponding variant declaration missing** Within a call to the procedure NEW or to the function SIZEOF, the indicated tag field specification fails to correspond to a variant within the associated record variable; or, the associated variable was not of a record type.
53	**Notify compiler maintenance group** If this error should occur, then notify the Pascal/VS compiler maintenance group at IBM. This is a compiler error.
54	**Expression must be numeric** Expressions which are prefixed with a sign ('+' or '-') must be of a type that is compatible with INTEGER or REAL. This also applies to expressions which are operands of such predefined functions as ABS and SQR.
55	**Expression must be of type real** The indicated call to ROUND or TRUNC has an argument (actual parameter) of an incorrect type. The predefined functions TRUNC and ROUND require an expression of type REAL as a parameter.
56	**Expression must be of type integer** The indicated expression must be of a type that is compatible with INTEGER.
57	**Parameter type does not match formal parameter** Within a procedure or function call, an expression or variable is being passed as an actual parameter which is of a type that is not compatible with the corresponding formal parameter.
58	**This expression must be a variable** An erroneous attempt was made to pass a non-variable as an actual parameter to a routine which expects a pass-by-**var** parameter.
59	**Number of parameters does not agree** Within a procedure or function call, the number of parameters being passed does not correspond with the number required.
60	**'(' expected**
61	**Constant expected**

62 **Type specification expected**

At the place indicated, a type definition is expected but is missing.

63 **'..' expected**

64 **Expression's type is incorrect or incompatible within context**

This error is caused by a number of reasons:

- A unary or binary operator is being applied to an expression which is of a type that is not valid for the operator.
- Two expressions being joined by a binary operator are of incompatible types.
- The parameters of the MIN/MAX functions are not of consistent types.
- Members of a set constructor have inconsistent types.

65 **Subrange lower bound > upper bound**

66 **Assignment to ptr qualified variant record invalid**

The indicated statement attempts to assign to the whole of a pointer qualified record with variant fields. Such an assignment is not valid under Pascal/VS. This restriction is necessary because the pointer qualified record may have been allocated with a size that is specific to its active variant.

Example of violation:

```
type
  R = record
        case BOOLEAN of
         TRUE: (C:CHAR);
         FALSE: (A: ALPHA)
       end;
var P : ->R;
   RR : R;
begin
  NEW(P,TRUE);
  P-> := RR     (*<===invalid assignment*)
end
```

67 **Real type not valid here**

The indicated expression is of type REAL. An expression of this type is not valid within the associated context.

68 **"OF" expected**

69 **Tag constant does not match tag field type**

Within a **record** definition, a variant tag is being defined which is of a type that is not compatible with the corresponding tag field type.

Within a call to NEW or SIZEOF, a tag value is specified which is of a type that is not compatible with the corresponding tag field type of an associated record variable.

70 **Duplicate variant field**

Within a **record** definition, a variant tag is being defined more than once.

71	**Not applicable to "PACKED" qualifier** The indicated type definition was qualified with the word **"packed"**. Such a qualification within the associated context is not valid.
72	**'[' expected**
73	**Array has too many elements** The length of the indicated array definition exceeds the addressability of the computer.
74	**']' expected**
76	**File of files not supported**
77	**Illegal reference to function name** The indicated identifier is the name of a function. It is being used in a way that is incorrect.
78	**Subscript type not compatible with index type** The indicated subscript expression is not of a type that is compatible with the declared subscript type for the array.
79	**Associated variable must be of a record type.** A variable associated with the indicated statement or expression is required to be of a record type according to context; but such is not the case.
80	**Record field qualifier not defined** The indicated record field does not exist for the associated record.
81	**Notify compiler maintenance group** If this error should occur, then notify the Pascal/VS compiler maintenance group at IBM. This is a compiler error.
82	**Associated variable must be of a pointer or file type** The indicated arrow qualified variable is not of a pointer or file type.
83	**Set element out of range** The indicated set member of a set constructor exceeds the allowed range for the set.
84	**Expression must be of a set type** The indicated expression is required to be of a set type in the context in which it is being used.
85	**Must be positive integer constant** The indicated expression fails to evaluate to a positive integer constant, which is required in the context in which it is being used.
86	**LEAVE/CONTINUE not within loop** The indicated **leave** or **continue** statement fails to reside within a loop construct.

87	**':=' expected**
89	**Jump out of procedure not supported** The target label of a **goto** statement must be local to the routine in which the statement resides. This is a Pascal/VS restriction.
90	**Label not declared** The indicated label did not appear in a **label** declaration.
92	**"THEN" expected**
93	**Redundant case alternative** The indicated **case** statement label is equal to a previous label within the same **case** statement.
95	**"UNTIL" expected**
96	**"DO" expected**
97	**FOR-loop index must be simple local variable** A **for**-loop variable must be declared as a simple automatic (**var**) variable, local to the routine in which the **for** loop resides. The indicated **for**-loop variable did not meet this criteria.
98	**"TO" expected**
99	**Label previously defined** The indicated label identifier was previously defined within the associated routine.
100	**Notify compiler maintenance group** If this error should occur, then notify the Pascal/VS compiler maintenance group at IBM. This is a compiler error.
101	**Notify compiler maintenance group** If this error should occur, then notify the Pascal/VS compiler maintenance group at IBM. This is a compiler error.
102	**Notify compiler maintenance group** If this error should occur, then notify the Pascal/VS compiler maintenance group at IBM. This is a compiler error.
103	**Expression must be of type BOOLEAN** The indicated expression which is associated with an **if**, **assert**, **while**, or **repeat** statement is required to represent a condition. Conditional expressions are of type BOOLEAN. The indicated expression failed to meet this criteria.
104	**Constant out of range** The indicated constant expression evaluated to a value which is outside the required range of its context.
105	**Identifier was previously declared** The indicated identifier within a declaration was previously declared within the same lexical scope.

106	**Undeclared identifier** The indicated identifier being referenced was not declared.
107	**Identifier is not in proper context** The indicated identifier is being used in a way that is not consistent with how it was declared.
108	**Notify compiler maintenance group** If this error should occur, then notify the Pascal/VS compiler maintenance group at IBM. This is a compiler error.
109	**Case label tag of wrong type** The value of the indicated **case** statement label is not of a type that is conformable to the **case** statement indexing expression.
110	**Loop will never execute** The indicated **for** loop will not execute at runtime. The compiler has determined that the terminating condition for the loop is unconditionally true.
111	**Loop range exceeds range of index** The indexing variable used for the indicated **for** loop was declared with a subrange that does not include the range indicated by the initial and final index values.
112	**'PROGRAM' header missing**
113	**Pending comment not terminated** A comment starting symbol was detected within a pending comment.
114	**Percent "%" statement not found** A '%' symbol was detected, but with no identifier following.
115	**Percent "%" identifier not recognized** A identifier following the '%' symbol is not recognized as a valid compiler directive.
116	**"ON" or "OFF" expected**
117	**Unrecognizable option in "%CHECK"**
120	**String constant requires truncation** The indicated string constant, which is being assigned to a variable or being passed to a routine, requires truncation because of its excessive length. Implicit truncation of strings is not permitted.
122	**"OTHERWISE" clause without associated CASE statement** The indicated **otherwise** statement is not within the context of a **case** statement.
123	**Maximum string length exceeded** The indicated expression produced a varying length string which exceeds 255 characters in length. 255 is the maximum allowed length for a varying length string.

125	**Real to integer conversion not valid** The indicated expression is of type real, but according to its context, it is required to be of type integer. Implicit real to integer conversion is not performed.
126	**Types not conformable in assignment** The indicated assignment statement attempts to assign an expression of a particular type to a variable of an incompatible type.
127	**File variable assignment not permitted** The left side of the indicated assignment statement is a variable of a file type. Assignment to file variables is not permitted.
128	**Not compile-time computable** The indicated expression fails to be a constant expression that can be evaluated at compile time.
129	**Assignment to "CONST" parameter invalid** The indicated variable declared as a formal **const** parameter within a particular routine may not be modified by an assignment.
130	**Assignment to FOR-loop index invalid** The indicated variable that is being used as a **for** loop index may not be modified by an assignment within the **for** loop statement.
131	**Passing "CONST" parameter by VAR invalid** The indicated variable declared as a formal **const** parameter may not be modified by being passed as an actual **var** parameter to a routine.
132	**Passing FOR-loop index by VAR invalid** The indicated variable that is being used as a **for** loop index may not be modified by being passed as an actual **var** parameter to a routine.
133	**Refer-back tagfield must not be typed** The indicated tag field specification within a record definition was found to reference a previous field within the record. Such refer-back references may not contain a type reference.
137	**Passing packed record field by VAR not valid** The indicated field of a packed record may not be passed as an actual **var** parameter to a routine.
138	**Passing SPACE component by VAR not valid** The component of a **space** variable may not be passed as an actual **var** parameter to a routine.
139	**Passing packed array element by VAR not valid** An element of a packed array variable may not be passed as an actual **var** parameter to a routine.
140	**Scalar PACKing does not match corresponding VAR parameter** The indicated variable that is being passed as a **var** parameter is of a compatible type, but has a different length than the corresponding formal parameter. This was caused by one being **packed** and the other unpacked.

142	**Must be an array variable** The indicated variable is required to be of an array type, but such is not the case.
143	**Offset qualified field not on proper boundary** The indicated field in a record definition is qualified with an offset which is not consistant with the boundary requirement of the field's type.
144	**Offset qualification value is too small** The indicated field in a record definition is qualified with an offset which causes an overlap with a previous field within the record.
145	**Type must be CHAR or PACKED ARRAY OF CHAR** The indicated expression is required by its context to be of type CHAR or **packed array[1..n] of** CHAR.
146	**Variables of type POINTER are not permitted** The special type 'POINTER' may only be applied to a formal parameter of a routine.
147	**Identifier was not declared as function** The indicated identifier is used as though it is a function name, but is not declared as such.
148	**Missing period '.' assumed**
149	**Not a valid comparison operation** The indicated expression performs a comparison operation on two entities for which such comparison is not allowed. Except for strings, variables of structured types may not be directly compared with each other. The only valid comparison operators for sets are '=', '<>', '<=', and '>='.
150	**ENTRY routines must be at the outermost nesting level** A routine declared as an ENTRY may not be nested within another routine.
151	**Fixed Point overflow or divide-by-zero** An integer expression consisting of constant operands causes a program error to occur when it is evaluated.

152	**Checking error will inevitably occur at execution time**
	This error indicates that the compiler has detected a condition related to a particular construct which will cause an execution time error.
	This error may occur at an assignment or at a routine call in which parameters are passed. It indicates that the range of the source expression (a scalar) does not overlap the declared range of the target. For example, the following assignment would cause this error to occur:
	`var I: 1..10;`
	`      J: 10..20;`
	`         ...`
	`   I := J+1; (*target's range: 1..10; source's range: 11..21 *)`

153	**LBOUND/HBOUND dimension number is invalid for variable**
154	**Low bound of subscript range is too large in magnitude** The indicated array definition has an illegal subscript range which causes addressing code to be outside the range of the target machine's capability.
155	**The ORD of all SET members must lie within 0..255** The ordinal value of any valid set member may not be less than 0 nor greater than 255.
156	**Length fields not applicable to non-TEXT files** A non-text file READ or WRITE contains a length qualified parameter. Length specifications have no meaning in non-text file I/O.
157	**STRING variable is smaller than file component** The error occurs when an attempt is made to perform a READ operation from a **file of** STRINGs into a string variable in which truncation is possible. The string variable must be declared with at least the same length as the file component.
158	**Routines passed as parameter must be at outermost nesting level** An attempt is being made to pass a routine as a parameter, but the routine being passed is nested within another. As a Pascal/VS restriction, routines being passed as parameters must not be nested within another routine.
159	**Recursive type reference is semantically incorrect** The compiler detected a degenerate **type** declaration of one of the following forms: I. **type** X = X; II. **type** X = ->X; III. **type** X = **record** ... F: X; ... **end**

160	**This SET operation will always produce the NULL set**
	Two disjoint sets are being intersected. The result will always be the null set [ ]. For example,
	var S1: **set of** 0..10;
	S2: **set of** 11..20;
	S3: **set of** 0..20;
	begin
	...
	S3 := S1 * S2; (* <== always produces the NULL set *)
	...
	end

161	**ELSE clause without associated IF statement**
162	**Must be an unPACKED array** The indicated array variable is erroneously declared as **packed** when the context requires it to be unpacked.
163	**Must be a PACKED array** The indicated array variable should have been declared as **packed**, but was not.

164	**Unrecognizable procedure/function directive** The indicated identifier was interpreted as a procedure or function directive but was not recognizable. The following are the only recognizable directives: - FORWARD - EXTERNAL - FORTRAN - ENTRY
165	**FORTRAN subroutines may not be passed as parameters** Only Pascal/VS routines may be passed as parameters; FORTRAN subroutines may not. One way to get around this problem is to define a Pascal/VS procedure which does nothing more than call the FORTRAN subroutine. The Pascal/VS procedure would then be passed in place of the FORTRAN subroutine.
166	**FORTRAN subroutine parameters may not be passed by value** All formal parameters of a FORTRAN subroutine must be passed by reference: either by **var** or by **const**.
167	**FORTRAN functions may return only scalar values** A FORTRAN function may only return values that are scalars (including floating point).
168	**%INCLUDE member not found in library** The library member which was to be included into the source program could not be found.
169	**Floating point computational error** The indicated floating point expression causes a program error when evaluated.
170	**Data storage exceeds addressability of machine** The memory required to contain all declared variables within a routine or main program exceeds the capacity of the computer; that is, it exceeds 16 megabytes.
171	**Only STATIC/DEF variables may be initialized** The only class of variables which may be initialized at compile time are **def** and **static** variables.
172	**Variable's address is not compile-time computable** The indicated **value** assignment could not be performed. In order for a variable to be initialized at compile-time, its address must be compile time computable.
173	**Array structure has too many elements** The indicated array structure contains more elements than was declared for the array type.
174	**Repetition factor applicable to constants only** Within a array structure, only a constant may be qualified with a repetition factor; a general expression may not.
175	**No corresponding record field** The indicated record structure contains more elements than there are fields within the record type.

176 **This identifier is a reserved name**

An attempt was made to declare an identifier which is a reserved name.

177 **Numeric labels must lie within the range 0..9999.**

178 **Identifier was previously referenced illegally**

The indicated identifier that was just declared was referenced previously within the associated routine. Pascal/VS requires an identifier to be declared prior to its use.

179 **Recursive reference within constant declaration**

A constant declaration of one of the following forms was detected:

```
  const X = X;
     or
  const X = "some expression involving X"
```

Such recursion within a constant declaration is not permitted.

180 **Repetition factor not applicable to record structures**

The indicated record structure contains a component which is qualified with a repetition factor. Only array structures are permitted to have repetition factors.

181 **Label previously referenced from a GOTO invalidly**

The indicated label was previously referenced in a **goto** statement that is not a constituent of the statement sequence in which the label is defined.

Example

```
  begin
    goto LABEL1;
    for I := 1 to 10 do
      begin
 LABEL1: A[I] := 0; (*<==label was previously referenced invalidly*)
       ...
      end;
  end
```

182 **A GOTO may not reference a label within a separate stmt sequence**

The indicated **goto** statement references a label which was previously defined within a statement sequence of which the **goto** is not a constituent. Such a reference is not permitted.

Example

```
  begin
    for I := 1 to 10 do
      begin
 LABEL1: A[I] := 0;
       ...
      end;
    goto LABEL1; (*<==invalid reference of label *)
  end
```

183 **CASE label outside range of indexing expression**

The indicated **case** label within a **case** statement has a value which is outside the range of the indexing expression. For example,

```
var I: 0..10;
begin
  case I*2 of (*range of index is 0..20 *)
   0: ...
   1..20: ...
   30: ...  (*<== this label is out of range of index*)
  end
end
```

184 **Second operand of MOD operation must be positive integer**

The indicated expression involving the **mod** operator was found to be invalid; the second operand is required to be a positive integer.

600 **Identifier used in type definition at line nnn is out of context: xxxx**

The identifier 'xxxx' appeared in a pointer type definition of the form '->xxxx' at line 'nnn', but the identifier was subsequently declared as something other than a type.

Example:

```
type X = ->Y;
   ...
var Y: INTEGER;  (* <=== would cause error 600 to be generated *)
```

601 **Type identifier referenced at line nnn is undeclared: xxxx**

The identifier 'xxxx' appeared in a pointer type definition of the form '->xxxx' at line 'nnn', but the identifier was not subsequently declared.

602 **Label xxxx was declared and/or referenced but was not defined**

The label named 'xxxx' was declared and/or referenced from within the associated routine, but was not ever defined.

603 **procedure/function xxxx was forwarded but not resolved**

The procedure or function named 'xxxx' was declared with the directive 'FORWARD', but the body of the routine was not subsequently declared.

16.2 INPUT/OUTPUT MESSAGES

No.	Message and Explanation
AMPX001I	**File could not be opened: ddname** An error occurred when an attempt was made to open the file whose DDNAME is 'ddname'. The most probable cause of this error is a missing ddname definition.
AMPX002I	**LRECL size too small for file ddname** The logical record length of the file with indicated ddname is not large enough to contain the data in one file component.
AMPX003I	**File is not open for output: ddname** An output operation was attempted on a file open for input.
AMPX004I	**File is not open for input: ddname** An input operation was attempted on a file open for output.
AMPX005I	**File has small format V record: ddname** The logical record length of a particular record within a variable record length file was too small to contain the file's component data.
AMPX006I	**Data larger than lrecl for file: ddname**
AMPX007I	**Invalid options in OPEN for file ddname** The options string passed to the OPEN procedure contains unrecognizable directives.
AMPX008I	**Missing member in file: member library** The indicated member could not be found in the partitioned data set.
AMPX009E	**Floating point overflow/underflow** The floating point number read by procedure READ was either too large or too small to be represented within the machine.

16.3 MEMORY MANAGEMENT MESSAGES

No.	Message and Explanation
AMPX050I	**Operand of RELEASE does not correspond to last MARK** The parameter passed to RELEASE did not have the value returned by the last call to MARK.
AMPX051I	**Operand of DISPOSE not allocated with NEW** A DISPOSE operation was attempted for a pointer which did not have a valid value as would have been returned by NEW.
AMPX053I	**Operand of DISPOSE already deallocated** An attempt was made to perform a DISPOSE operation on a pointer which referenced heap storage which had been previously released.

16.4 MATH PACKAGE MESSAGES

No.	Message and Explanation
AMPX100I	LN: argument <= 0.0. The natural logarithm function (LN) was called with a 0 or negative argument.
AMPX101I	SQRT: argument < 0.0, zero returned as result The square root function (SQRT) was called with a negative argument.
AMPX102I	EXP: argument too large, exceeds 174.67309 The argument of the EXP function is too large; the result of the call exceeds the largest real number that can be represented: 7.237e+75.
AMPX103I	RANDOM: seed is out of range. The function RANDOM was called with an argument which is either negative or greater than 1048575 (which is the allowed maximum).
AMPX104I	SIN/COS: argument too large exceeds (pi/2)**50. A call to SIN or COS was made with an argument that is too large for an accurate result to be computed.

16.5 MESSAGES FROM PASCALVS EXEC

The following messages are given by the PASCALVS EXEC of CMS to indicate the status of the compiler invocation. They are shown below with their associated return codes. (A non-zero return code indicates a terminated compilation.)

RC	Message and Explanation
1	**File name is missing** The exec was invoked without specifying a file name.
2	**Unable to find 'fn' PASCAL** The specified file name could not be found.
16	**Unable to find the 'name' MACLIB** The specified maclib file could not be found.
32	**More than 8 maclibs specified** The maximum number of MACLIBS that may be specified when invoking the PASCALVS EXEC is eight.

APPENDIXES

A.0 COMMAND SYNTAX NOTATION

The syntax notation used to illustrate TSO commands is explained in the manual TSO Command Language Reference (GC28-0646). The notation used to illustrate CMS commands is explained in the manual VM/370: CMS Command and Macro Reference (GC20-1818).

Briefly, the conventions used by both notations are as follows.

- Items in brackets [ ] are optional. If more than one item appears in brackets, then no more than one of them may be specified; they are mutually exclusive.
- Items in capital letters are keywords. The command name and keywords must be spelled as shown.
- Items in lowercase letters must be replaced by appropriate names or values.
- Items which are underlined represent defaults.
- The special characters ' ( ) * must be included where shown.

This section describes how to install Pascal/VS under OS/VS2 and CMS-VM/370 from the distribution tape.

All VS2 partitioned data sets (other than the compiler source) were stored on the tape by using the IEBCOPY utility program. VS2 sequential data sets were stored by using the IEBGENER utility program.

The CMS version of the package is located at file 14 on the tape. It was stored by using the TAPE DUMP command.

The source of the compiler was stored using the utility program IEBUPDTE.

The files on the distribution tape contain the following data sets.

File 1: INSTALL.CNTL
A sample of the job control language (JCL) required to install Pascal/VS under OS/VS2 (MVS).

File 2: LOADSRC.CNTL
A sample of the job control language (JCL) required to load the Pascal/VS source from the distribution tape.

File 3: PASCALVS.CONTENTS
A sequential data set which lists the contents of the Pascal/VS package.

File 4: PASCALVS.LINKLIB
A partitioned data set which contains the modules of the compiler.

File 5: PASCALVS.LOAD
A partitioned data set which contains the Pascal/VS run time library.

File 6: PASDEBUG.LOAD
A partitioned data set which contains the Pascal/VS debug library.

File 7: PASCALVS.MACLIB
The standard include library.

File 8: PASCALVS.CLIST
A partitioned data set containing two clists: PASCALVS and PASCMOD.

File 9: PASCALVS.PROCLIB
A partitioned data set which contains the JCL cataloged procedures for running the compiler as a batch job under MVS.

File 10: MASTER.MENUS
A partitioned data set which contains SPF menus which will permit Pascal/VS to be invoked from the program product SPF.

File 11: MASTER.PROCS
A partitioned data set which contains the command procedures necessary to invoke Pascal/VS under the program product SPF.

File 12: PASCALVS.MESSAGES
A sequential data set which contains the compiler messages.

File 13: SAMPLE.PASCAL
A sample Pascal/VS program.

File 14: CMS dump of the entire Pascal/VS package:

- **PASCALVS CONTENTS**
  A listing of the contents of the Pascal/VS package.

- **PASCALL MODULE**
  The first pass of the compiler.

- **PASCALT MODULE**
  The second pass of the compiler.

- **PASCALVS TXTLIB**
  The Pascal/VS run time library.

- **PASDEBUG TXTLIB**
  The Pascal/VS debug library.

- **PASCALVS MACLIB**
  The standard %INCLUDE library.

- **PASCALVS EXEC**
  CMS EXEC which invokes the compiler

- **PASCAL1 EXEC**
  an internal EXEC invoked from PASCALVS EXEC.

- **PASCAL2 EXEC**
  an internal EXEC invoked from PASCALVS EXEC.

- **PASCMOD EXEC**
  CMS EXEC which creates a load module from a compiled Pascal/VS program.

- **PASCALVS MESSAGES**
  List of the compiler messages.

- **LOADSRC EXEC**
  An EXEC which will load the source of the compiler from the tape.

- **SAMPLE PASCAL**
  A sample program.

File 15: PASCALL.PASCAL

The source of the first pass of the compiler.

File 16: PASCALT.PASCAL
The source of the second pass of the compiler.

File 17: PASCALD.PASCAL
The source of the interactive debugger.

File 18: PASCALX.PASCAL
The source of the runtime library routines.

File 19: PASCALX.ASM
The source of the operating system interface routines.

File 20: MACLIBL.PASCAL
Include library for first pass of the compiler.

File 21: MACLIBT.PASCAL
Include library for second pass of the compiler.

File 22: MACLIBD.PASCAL
Include library for interactive debugger.

File 23: MACLIBX.PASCAL
Include library for runtime routines.

B.1 INSTALLING PASCAL/VS UNDER CMS

To install Pascal/VS under CMS perform the following:

1. Have the distribution tape mounted at address 181.

2. Link to the mini-disk (in write mode) where the compiler is to be stored. This is done with the CP LINK command. The mini-disk must have at least 1210 blocks of free storage[10].

3. Access this disk with the ACCESS command.

4. Execute the following two commands:

   ```
   TAPE FSF 13
   TAPE LOAD * * m
   ```

   where "m" is the single letter file mode of the disk that was accessed in the previous step.

[10] 800 byte blocks are assumed. This amount is equivalent to 5 cylinders on a 3330 disk.

```
//JOBNAME JOB ,REGION=50K
//STEP1  EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=*
//SYSUT1 DD DSN=PASCALVS.INSTALL.CNTL,
//          VOL=SER=TAPEVOL,
//          UNIT=TAPE,LABEL=(1,NL),
//          DCB=(LRECL=80,RECFM=FB,BLKSIZE=3120,DEN=3),
//          DISP=OLD
//SYSUT2 DD DSN=XXXXXXXX.INSTALL.CNTL,DISP=(NEW,CATLG),
//          DCB=(LRECL=80,RECFM=FB,BLKSIZE=3120),
//          UNIT=3330,VOL=SER=DISKVOL,
//          SPACE=(TRK,(1,1))
//SYSIN  DD DUMMY
```

Figure 70. Sample JCL to retrieve first file of distribution tape.

B.2 INSTALLING PASCAL/VS UNDER VS2

This section explains how to install Pascal/VS under an OS/VS2 system.

B.2.1 Loading Files from Distribution Tape

A sample of the job control language required to install Pascal/VS under VS2 (MVS) is stored as the first file of the distribution tape. To retrieve this data set, the utility program IEBGENER must be used. The JCL shown in Figure 70 may serve as a model job to retrieve this file. DD operands which are high-lighted will require modification to suit your installation requirements. The serial number of the distribution tape must be placed where the name "TAPEVOL" appears in the DD card named SYSUT1.

The data set name (DSN=) in the DD card named SYSUT2 is arbitrary. It is the name of the data set where the first file on the tape is to be stored. The appropriate UNIT and volume serial number for disk storage must be specified for DD SYSUT2.

Figure 71 on page 122, Figure 72 on page 123, and Figure 73 on page 124 contain a listing of the first file of the distribution tape. The following modifications are required prior to submitting this job.

- The name "TAPEVOL" must be replaced with the volume serial number of the distribution tape in the DD statement named SYSUT1 in job step STEP1.
- The UNIT specification for tapes has been given the generic name of "TAPE"; this should be changed to the appropriate generic at your installation.
- The UNIT specification for disk storage has been specified as "3330"; this should be changed to the appropriate specification at your installation.
- The disk volume on which Pascal/VS is to be installed must be specified where indicated ("DISKVOL") in the following DD statements:
  in STEP1: SYSUT2
  in STEP2: SYSUT2
  in STEP3: DS4, DS5, DS6,
  DS7, DS8, DS9,
  DS10, DS11
  in STEP4: SYSUT2
  in STEP5: SYSUT2
- The DD statements named SYSUT3 and SYSUT4 in job step STEP3 represent temporary work storage. The generic name "SYSDA" is used as a UNIT specification; this should be changed to the appropriate generic at your installation.
- The tape density is specified within the DEN suboperand of the DCB attributes. In the sample job, DEN is set to 3 which indicates a tape density of 1600 BPI. If your distribution tape is at some other density, then the DEN operands should be changed accordingly.
- The high level qualifier of data set names that are to be cataloged should be modified to follow installation conventions. (The examples in this manual assume a high level qualifier of "SYS1".)

```
//INSTALL  JOB ,REGION=128K
//*
//*    FILE 2 -- SOURCE INSTALLATION JOB
//*
//STEP1  EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=*
//SYSUT1 DD DSN=LOADSRC.CNTL,
//          VOL=(,RETAIN,SER=TAPEVOL),
//          UNIT=TAPE,LABEL=(2,NL),
//          DCB=(LRECL=80,RECFM=FB,BLKSIZE=3120,DEN=3),
//          DISP=(OLD,PASS)
//SYSUT2 DD DSN=SYS1.LOADSRC.CNTL,DISP=(NEW,CATLG),
//          DCB=(LRECL=80,RECFM=FB,BLKSIZE=3120),
//          UNIT=3330,VOL=SER=DISKVOL,
//          SPACE=(3120,(1,1))
//SYSIN  DD DUMMY
//*
//*    FILE 3 -- PASCALVS CONTENTS
//*
//STEP2  EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=*
//SYSUT1 DD DSN=PASCALVS.CONTENTS,
//          VOL=REF=*.STEP1.SYSUT1,
//          UNIT=TAPE,LABEL=(3,NL),
//          DCB=(LRECL=80,RECFM=VB,BLKSIZE=3120,DEN=3),
//          DISP=(OLD,PASS)
//SYSUT2 DD DSN=SYS1.PASCALVS.CONTENTS,DISP=(NEW,CATLG),
//          DCB=(LRECL=80,RECFM=VB,BLKSIZE=3120),
//          UNIT=3330,VOL=SER=DISKVOL,
//          SPACE=(3120,(1,1))
//SYSIN  DD DUMMY
//*
//*  FILE 4  -- PASCALVS.LINKLIB
//*  FILE 5  -- PASCALVS.LOAD
//*  FILE 6  -- PASDEBUG.LOAD
//*  FILE 7  -- PASCALVS.MACLIB
//*  FILE 8  -- PASCALVS.CLIST
//*  FILE 9  -- PASCALVS.PROCLIB
//*  FILE 10 -- SPF.MASTER.MENUS
//*  FILE 11 -- SPF.MASTER.PROCS
//*
//STEP3  EXEC PGM=IEBCOPY
//DS4    DD DSN=SYS1.PASCALVS.LINKLIB,DISP=(NEW,CATLG),
//          DCB=(BLKSIZE=13030,RECFM=U,DSORG=PO),
//          UNIT=3330,VOL=SER=DISKVOL,
//          SPACE=(TRK,(70,10,3))
//FILE4  DD DSN=PASCALVS.LINKLIB,
//          VOL=REF=*.STEP1.SYSUT1,
//          UNIT=TAPE,LABEL=(4,NL),
//          DCB=BLKSIZE=13030,
//          DISP=(OLD,PASS)
//DS5    DD DSN=SYS1.PASCALVS.LOAD,DISP=(NEW,CATLG),
//          DCB=(BLKSIZE=13030,RECFM=U,DSORG=PO),
//          UNIT=3330,VOL=SER=DISKVOL,
//          SPACE=(TRK,(14,10,36))
//FILE5  DD DSN=PASCALVS.LOAD,
//          VOL=REF=*.STEP1.SYSUT1,
//          DCB=BLKSIZE=13030,
//          UNIT=TAPE,LABEL=(5,NL),
//          DISP=(OLD,PASS)
//DS6    DD DSN=SYS1.PASDEBUG.LOAD,DISP=(NEW,CATLG),
//          DCB=(BLKSIZE=13030,RECFM=U,DSORG=PO),
//          UNIT=3330,VOL=SER=DISKVOL,
//          SPACE=(TRK,(8,1,7))
```

Figure 71. Sample installation job: (continued in Figure 72 on page 123)

```
//FILE6  DD DSN=PASDEBUG.LOAD,
//          VOL=REF=*.STEP1.SYSUT1,
//          DCB=BLKSIZE=13030,
//          UNIT=TAPE,LABEL=(6,NL),
//          DISP=(OLD,PASS)
//DS7    DD DSN=SYS1.PASCALVS.MACLIB,DISP=(NEW,CATLG),
//          DCB=(BLKSIZE=3120,RECFM=FB,LRECL=80,DSORG=PO),
//          UNIT=3330,VOL=SER=DISKVOL,
//          SPACE=(TRK,(7,2,3))
//FILE7  DD DSN=PASCALVS.MACLIB,
//          VOL=REF=*.STEP1.SYSUT1,
//          UNIT=TAPE,LABEL=(7,NL),
//          DCB=BLKSIZE=3120,
//          DISP=(OLD,PASS)
//DS8    DD DSN=SYS1.PASCALVS.CLIST,DISP=(NEW,CATLG),
//          DCB=(BLKSIZE=3120,RECFM=VB,LRECL=255,DSORG=PO),
//          UNIT=3330,VOL=SER=DISKVOL,
//          SPACE=(TRK,(4,2,5))
//FILE8  DD DSN=PASCALVS.CLIST,
//          VOL=REF=*.STEP1.SYSUT1,
//          DCB=BLKSIZE=3120,
//          UNIT=TAPE,LABEL=(8,NL),
//          DISP=(OLD,PASS)
//DS9    DD DSN=SYS1.PASCALVS.PROCLIB,DISP=(NEW,CATLG),
//          DCB=(BLKSIZE=3120,RECFM=FB,LRECL=80,DSORG=PO),
//          UNIT=3330,VOL=SER=DISKVOL,
//          SPACE=(TRK,(4,2,2))
//FILE9  DD DSN=PASCALVS.PROCLIB,
//          VOL=REF=*.STEP1.SYSUT1,
//          UNIT=TAPE,LABEL=(9,NL),
//          DCB=BLKSIZE=3120,
//          DISP=(OLD,PASS)
//DS10   DD DSN=SYS1.MASTER.MENUS,DISP=(NEW,CATLG),
//          DCB=(BLKSIZE=3120,RECFM=VB,LRECL=84,DSORG=PO),
//          UNIT=3330,VOL=SER=DISKVOL,
//          SPACE=(TRK,(13,2,6))
//FILE10 DD DSN=MASTER.MENUS,
//          VOL=REF=*.STEP1.SYSUT1,
//          UNIT=TAPE,LABEL=(10,NL),
//          DCB=BLKSIZE=3120,
//          DISP=(OLD,PASS)
//DS11   DD DSN=SYS1.MASTER.PROCS,DISP=(NEW,CATLG),
//          DCB=(BLKSIZE=3120,RECFM=VB,LRECL=84,DSORG=PO),
//          UNIT=3330,VOL=SER=DISKVOL,
//          SPACE=(TRK,(1,1,2))
//FILE11 DD DSN=MASTER.PROCS,
//          VOL=REF=*.STEP1.SYSUT1,
//          UNIT=TAPE,LABEL=(11,NL),
//          DCB=BLKSIZE=3120,
//          DISP=(OLD,PASS)
//SYSPRINT  DD SYSOUT=*
//SYSUT3    DD UNIT=SYSDA,SPACE=(TRK,(1))
//SYSUT4    DD UNIT=SYSDA,SPACE=(TRK,(1))
//SYSIN     DD *
  COPY OUTDD=DS4,INDD=FILE4
  COPY OUTDD=DS5,INDD=FILE5
  COPY OUTDD=DS6,INDD=FILE6
  COPY OUTDD=DS7,INDD=FILE7
  COPY OUTDD=DS8,INDD=FILE8
  COPY OUTDD=DS9,INDD=FILE9
  COPY OUTDD=DS10,INDD=FILE10
  COPY OUTDD=DS11,INDD=FILE11
/*
```

Figure 72. Sample installation job: (continued in Figure 73 on page 124)

```
//*
//*    FILE 12-- PASCALVS MESSAGES
//*
//STEP4  EXEC PGM=IEBGENER
//SYSPRINT   DD SYSOUT=*
//SYSUT1 DD DSN=PASCALVS.MESSAGES,
//          VOL=REF=*.STEP1.SYSUT1,
//          UNIT=TAPE,LABEL=(12,NL),
//          DCB=(LRECL=80,RECFM=VB,BLKSIZE=3120,DEN=3),
//          DISP=(OLD,PASS)
//SYSUT2 DD DSN=SYS1.PASCALVS.MESSAGES,DISP=(NEW,CATLG),
//          DCB=(LRECL=80,RECFM=VB,BLKSIZE=3120),
//          UNIT=3330,VOL=SER=DISKVOL,
//          SPACE=(TRK,(1,1))
//SYSIN  DD DUMMY
//*
//*    FILE 13-- SAMPLE PASCAL
//*
//STEP5  EXEC PGM=IEBGENER
//SYSPRINT   DD SYSOUT=*
//SYSUT1 DD DSN=SAMPLE.PASCAL,
//          VOL=REF=*.STEP1.SYSUT1,
//          UNIT=TAPE,LABEL=(13,NL),
//          DCB=(LRECL=80,RECFM=FB,BLKSIZE=3120,DEN=3),
//          DISP=(OLD,KEEP)
//SYSUT2 DD DSN=SYS1.SAMPLE.PASCAL,DISP=(NEW,CATLG),
//          DCB=(LRECL=80,RECFM=FB,BLKSIZE=3120),
//          UNIT=3330,VOL=SER=DISKVOL,
//          SPACE=(TRK,(1,1))
//SYSIN  DD DUMMY
```

Figure 73. Sample installation job: (continued from Figure 71 on page 122 and Figure 72)

B.2.2 The TSO Clists

Distributed with the compiler are two CLISTs: PASCALVS and PASCMOD. These CLISTs reside in the partitioned data set PASCALVS.CLIST (file 8 of the distribution tape).

These CLISTs should be stored in a public CLIST library that is accessable to TSO users through DDname SYSPROC.

Each CLIST must be modified so that the correct high level qualifier name is used to reference the Pascal/VS data sets. In PASCALVS, the symbol named "FIRSTNAME" should be set to the appropriate name. In PASCMOD, the symbols named "LIBRARY" and "DEBUGLIB" should be set to the names of the Pascal/VS run time library and the debug library, respectively.

B.2.3 Cataloged Procedures

Distributed with the compiler are four cataloged procedures for invoking the compiler from a batch job: PASCC, PASCCG, PASCCL, and PASCCLG. These procedures reside in the partitioned data set PASCALVS.PROCLIB (file 9 of the distribution tape).

These procedures should be stored in a cataloged procedure library, so that the names will be recognized when referenced from a batch job.

Each procedure must be customized to reflect the data set naming convention chosen at your installation. For a listing of the cataloged procedures see "IBM Supplied Cataloged Procedures" on page 21.

B.2.4 SPF Menus and Procedures

If your TSO installation utilizes the Structured Programming Facility (SPF) (program number 5740-XT8), you may invoke the Pascal/VS compiler from SPF by means of the foreground/background menus.

File 11 on the distribution tape is a partitioned data set which contains the SPF menus required to add Pascal/VS to the list of compilers which may be invoked in the foreground/background menu of SPF. Each member in this data set should be copied into the partitioned data set named[11]

"SPF22.MASTER.MENUS"

[11] At some installations this data set may be named "SPF22.MOD1.MENUS".

The following members of this data set will be replaced:[12]

```
FORA
JOBA
JOBB
```

All other members will be new.[13]

File 11 of the tape is a partitioned data set which contains the foreground and background procedures for invoking the compiler. Each member of this data set should be placed in the data set named[14]

"SPF22.MASTER.PROCS"

The primary option menu of SPF is the member named APRIOPT in SPF22.-MASTER.MENUS. This menus should be modified so that the selection "5.9" will activate the Pascal/VS foreground menu, and the selection "4.7" will activate the Pascal/VS background menu. For information on installing and customizing SPF refer to the manual TSO 3270 Display Support and Structured Programming Facility Version 2.2: Installation and Customization Guide(SH20-2402).

B.3 LOADING THE SOURCE UNDER CMS

The compiler source is stored on the distribution tape beginning at file 15; that is, 14 tape marks from the beginning of the tape. It consists of nine tape files stored in the IEBUPDTE format. To read such a format under CMS, the TAPPDS command must be utilized.

The LOADSRC EXEC, which is provided as part of the Pascal/VS package, may be used to load all of the source files to a single disk. To run this EXEC, perform the following:

1. Have the distribution tape mounted at address 181.
2. Access the disk where the source files are to be stored in R/W mode. The disk must have the equivalent of 45 free cylinders of 3330 storage.[15]
3. Make sure that there is the equivalent of at least 2 free cylinders of 3330 storage on your "A" disk.
4. Invoke the LOADSRC EXEC as follows:

   LOADSRC fm

   where "fm" is the single letter file mode of the disk to where the source files are to be placed. The EXEC will print out messages as it processes the tape.

B.4 LOADING THE SOURCE UNDER VS2

The compiler source is stored on the distribution tape beginning at file 15. It consists of nine tape files stored in the IEBUPDTE format.

File 2 of the distribution tape contains the JCL which copies the source files to disk storage. This file is unloaded when the compiler is installed and has been given the name "LOADSRC.CNTL".

Prior to submitting the job, it must be customized as follows:

- In ddname SYSIN of jobstep STEP1, the volume serial number of the distribution tape should be placed where the name TAPEVOL is shown.
- The UNIT specification for tapes has been given the generic name "TAPE"; this should be changed to the appropriate generic at your installation.
- The UNIT specification for disk storage has been specified as "3330"; this should be changed to the appropriate specification at your installation.
- The disk volume on which the source files are to be stored must replace the name "DISKVOL" in the DD statement named SYSUT2 in each job step.
- The high level qualifier for the data set names to be cataloged is arbitrary. In the supplied JCL, the name "SOURCE" is used.
- The tape density is specified within the DEN suboperand of the DCB attributes. In the JCL, DEN is set to 3 which indicates a tape density of 1600 BPI. If your distribution tape is at some other density, then the DEN operands should be changed accordingly.

[12] As a precautionary measure, we suggest that you rename the members FORA, JOBA, and JOBB prior to replacing them with the new copy.

[13] You should look at the names of each member that we are supplying to make sure that they do not conflict with any previously existing member.

[14] At some installations this data set may be named "SPF22.MOD1.PROCS".

[15] This is roughly 15000 800-byte blocks. Once the source files have been installed, you may find it desirable to pack them in order to save disk storage.

```
//LOADSRC  JOB ,REGION=50K
//*
//*    FILE 15 -- PASCALL PASCAL - COMPILER SOURCE
//*
//STEP1 EXEC PGM=IEBUPDTE,PARM=NEW
//SYSUT2   DD DSN=SOURCE.PASCALL.PASCAL,DISP=(NEW,CATLG),
//            UNIT=3330,DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB),
//            VOL=SER=DISKVOL,SPACE=(TRK,(132,43,5))
//SYSIN    DD UNIT=TAPE,VOL=(,RETAIN,SER=TAPEVOL),LABEL=(15,NL),
//            DISP=(OLD,PASS),
//            DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DEN=3)
//SYSPRINT DD SYSOUT=*
//*
//*    FILE 16 -- PASCALT PASCAL  - TRANSLATOR SOURCE
//*
//STEP2 EXEC PGM=IEBUPDTE,PARM=NEW
//SYSUT2   DD DSN=SOURCE.PASCALT.PASCAL,DISP=(NEW,CATLG),
//            UNIT=3330,DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB),
//            VOL=SER=DISKVOL,SPACE=(TRK,(117,39,5))
//SYSIN    DD UNIT=TAPE,VOL=REF=*.STEP1.SYSIN,LABEL=(16,NL),
//            DISP=(OLD,PASS),
//            DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DEN=3)
//SYSPRINT DD SYSOUT=*
//*
//*    FILE 17 -- PASCALD PASCAL - DEBUG SOURCE
//*
//STEP3 EXEC PGM=IEBUPDTE,PARM=NEW
//SYSUT2   DD DSN=SOURCE.PASCALD.PASCAL,DISP=(NEW,CATLG),
//            UNIT=3330,DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB),
//            VOL=SER=DISKVOL,SPACE=(TRK,(33,9,5))
//SYSIN    DD UNIT=TAPE,VOL=REF=*.STEP1.SYSIN,LABEL=(17,NL),
//            DISP=(OLD,PASS),
//            DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DEN=3)
//SYSPRINT DD SYSOUT=*
//*
//*    FILE 18 -- PASCALX PASCAL - RUN TIME ENVIRONMENT SOURCE
//*
//STEP4 EXEC PGM=IEBUPDTE,PARM=NEW
//SYSUT2   DD DSN=SOURCE.PASCALX.PASCAL,DISP=(NEW,CATLG),
//            UNIT=3330,DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB),
//            VOL=SER=DISKVOL,SPACE=(TRK,(69,24,5))
//SYSIN    DD UNIT=TAPE,VOL=REF=*.STEP1.SYSIN,LABEL=(18,NL),
//            DISP=(OLD,PASS),
//            DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DEN=3)
//SYSPRINT DD SYSOUT=*
//*
//*    FILE 19 -- PASCALX ASM - RUN TIME ENVIRONMENT SOURCE
//*
//STEP5 EXEC PGM=IEBUPDTE,PARM=NEW
//SYSUT2   DD DSN=SOURCE.PASCALX.ASM,DISP=(NEW,CATLG),
//            UNIT=3330,DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB),
//            VOL=SER=DISKVOL,SPACE=(TRK,(16,1,4))
//SYSIN    DD UNIT=TAPE,VOL=REF=*.STEP1.SYSIN,LABEL=(19,NL),
//            DISP=(OLD,PASS),
//            DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DEN=3)
//SYSPRINT DD SYSOUT=*
```

Figure 74. Listing of the JCL to copy source files from tape: this job is stored as file 2 of the distribution tape. (continued in Figure 75 on page 127).

```
//*
//*    FILE 20 -- MACLIBL PASCAL  - %INCLUDE LIBRARY FOR COMPILER
//*
//STEP6 EXEC PGM=IEBUPDTE,PARM=NEW
//SYSUT2  DD DSN=SOURCE.MACLIBL.PASCAL,DISP=(NEW,CATLG),
//           UNIT=3330,DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB),
//           VOL=SER=DISKVOL,SPACE=(TRK,(21,7,4))
//SYSIN   DD UNIT=TAPE,VOL=REF=*.STEP1.SYSIN,LABEL=(20,NL),
//           DISP=(OLD,PASS),
//           DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DEN=3)
//SYSPRINT DD SYSOUT=*
//*
//*    FILE 21 -- MACLIBT PASCAL - %INCLUDE LIBRARY FOR TRANSLATOR
//*
//STEP7 EXEC PGM=IEBUPDTE,PARM=NEW
//SYSUT2  DD DSN=SOURCE.MACLIBT.PASCAL,DISP=(NEW,CATLG),
//           UNIT=3330,DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB),
//           VOL=SER=DISKVOL,SPACE=(TRK,(19,7,4))
//SYSIN   DD UNIT=TAPE,VOL=REF=*.STEP1.SYSIN,LABEL=(21,NL),
//           DISP=(OLD,PASS),
//           DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DEN=3)
//*
//*    FILE 22 -- MACLIBD PASCAL - %INCLUDE LIBRARY FOR DEBUG
//*
//STEP8 EXEC PGM=IEBUPDTE,PARM=NEW
//SYSUT2  DD DSN=SOURCE.MACLIBD.PASCAL,DISP=(NEW,CATLG),
//           UNIT=3330,DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB),
//           VOL=SER=DISKVOL,SPACE=(TRK,(2,1,1))
//SYSIN   DD UNIT=TAPE,VOL=REF=*.STEP1.SYSIN,LABEL=(22,NL),
//           DISP=(OLD,PASS),
//           DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DEN=3)
//SYSPRINT DD SYSOUT=*
//*
//*    FILE 23 -- MACLIBX PASCAL - %INCLUDE/MACRO LIBRARY FOR RUN TIME
//*                                 ENVIRONMENT
//*
//STEP9 EXEC PGM=IEBUPDTE,PARM=NEW
//SYSUT2  DD DSN=SOURCE.MACLIBX.PASCAL,DISP=(NEW,CATLG),
//           UNIT=3330,DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB),
//           VOL=SER=DISKVOL,SPACE=(TRK,(9,1,2))
//SYSIN   DD UNIT=TAPE,VOL=REF=*.STEP1.SYSIN,LABEL=(23,NL),
//           DISP=OLD,
//           DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DEN=3)
//SYSPRINT DD SYSOUT=*
```

Figure 75. Listing of the JCL to copy source files from tape: (continued from Figure 74)