SH20-6162-2

Program Offering

Pascal/VS

Programmer's Guide

Program Number: 5796-PNQ

Pascal/VS is a Pascal compiler operating in VS1, MVS and VM/CMS. Originally designed as a high level programming language to teach computer programming by Professor Nicklaus Wirth (circa 1968), Pascal has emerged as an influential and well accepted user language in today's data processing environment. Pascal provides the user with the ability to produce very reliable code by performing many error detection checks automatically.

The compiler adheres to the currently ANSI and ISO (Level 0) standard (with minor deviations) and includes many important extensions. The language extensions include: separate compilation, dynamic character strings and extended I/O capabilities. The implementation features include: fast compilation, optimization and a symbolic terminal oriented debugger that allows the user to debug a program quickly and efficiently.

This manual is a guide to the use of the computer in the VS1, MVS and VM/CMS operating environments.

PROGRAM SERVICES

During a specified number of months immediately following initial availability of each licensed program, the customer may submit documentation to the designated IBM location below when he/she encounters a problem which his/her diagnosis indicates is caused by a defect in the licensed program. During this period only, IBM, through the program sponsor(s), will, without additional charge, respond to an error in the current unaltered release of the licensed program by issuing known error correction information to the customer reporting the problem and/or issuing corrected or notice of availability of corrected code. However, IBM does not guarantee service results or represent or warrant that all errors will be corrected. Any onsite program services or assistance may be provided at a charge.

WARRANTY

THE LICENSED PROGRAM DESCRIBED IN THIS MANUAL IS DISTRIBUTED ON AN "AS IS" BASIS WITHOUT WARRANTY OF ANY KIND EITHER EXPRESSED OR IMPLIED.

Central Service Location: IBM Corporation
555 Bailey Ave.
P.O. Box 50020
San Jose, CA 95150
Attn: Luis Tan
IBM Tieline: 8/543-4392
Telephone: (408) 463-4392

Note: Non-US customers should contact their designated support group in their country.

Information concerning Program Services for this Program Offering can be found in Availability Notice G320-6387.

Third Edition (February 1985)

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available outside the United States.

A form for readers' comments has been provided at the back of this publication. If this form has been removed, address comments to: The Central Service Location. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation, 1980, 1981, and 1985.

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 Language Reference Manual, order number SH20-6168. 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.
- IBM Virtual Machine/Personal Computer User's Guide, order number SC24-5254. This manual describes describes the VM/PC operating system which runs on the IBM Personal Computer XT/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.

RELEASE 2.2

The following is a list of the functional changes that were made to Pascal/VS for Release 2.2.

- The interactive debugger now supports 32 breakpoints.
- Two new predefined constants have been added to the compiler: MINREAL and MAXREAL.
- The LANGLVL(STDRES) compiler option has been added to allow the user to use the non-standard Pascal/VS reserved words as identifiers.
- A new predefined function, ADDR, accepts a variable name and returns the location of that variable in storage.
- Structured array constants may now be passed as the source arrays to PACK and UNPACK.

RELEASE 2.1

The following is a list of the functional changes that were made to Pascal/VS for Release 2.1.

- A procedure (or function) at any nesting level may now be passed as a routine parameter. The previous restriction which required such procedures to be at the outermost nesting level of a module has been removed.
- Two new options may be applied to files when they are opened: UCASE and NOCC.
- Rules have been relaxed in passing fields of packed records by var to a routine.
- The "STACK" and "HEAP" run time options have been added to control the amount by which the stack and heap are extended when an overflow occurs.
- The syntax of a "structured constant" which contains non-simple constituents has been simplified.

RELEASE 2.0

The following is a list of the functional changes that were made to Pascal/VS for Release 2.0.

- Pascal/VS now supports single precision floating point (32 bit) as well as double precision floating point (64 bit).
- Files may be opened for updating with the UPDATE procedure.
- Files may be opened for terminal input (TERMIN) and terminal output (TERMOUT) so that I/O may take place directly to the user's terminal without going through the DDNAME interface.
- The MAIN directive permits you to define a procedure that may be invoked from a non-Pascal environment. A procedure that uses this directive is not reentrant.
- The REENTRANT directive permits you to define a procedure that may be invoked from a non-Pascal environment. A procedure that uses this directive is reentrant.
- A new predefined type, STRINGPTR, has been added that permits you to allocate strings with the NEW procedure whose maximum size is not defined until the invocation of NEW.

- A new parameter passing mechanism is provided that allows strings to be passed into a procedure or function without requiring you to specify the maximum size of the string on the formal parameter.
- The maximum size of a string has been increased to 32767 characters.
- The Pascal/VS compiler is now fully reentrant.
- Code produced from the compiler will be reentrant if static storage is not modified.
- Pascal/VS programs may contain source lines up to 100 characters in length.
- Files may be accessed based on relative record number (random access).
- Run time errors may be intercepted by the user's program.
- Run time diagnostics have been improved.
- Pascal/VS will flag extensions when the option "LANGLVL(STD)" is used.
- A mechanism has been provided so that Pascal/VS routines may be called from other languages.
- All record formats acceptable to QSAM are now supported by the Pascal/VS I/O facilities.
- A procedure or function may now be exited by means of the **goto** statement.
- You may now declare an array variable where each element of the array is a file.
- You may define a file to be a field of a record structure.
- Files may now be allocated in the heap (as a dynamic variable) and accessed via a pointer.
- You may now define a subrange of INTEGER which is allocated to 3 bytes of storage. Control over signed or unsigned values is determined by the subrange.
- Variables may be declared in the outermost scope of a SEGMENT. These variables are defined to overlay the variables in the outermost scope of the main program.
- The PDSIN procedure opens a member of a library file (partitioned dataset) for input.
- The PDSOUT procedure opens a member of a library file (partitioned dataset) for output.
- A procedure or function that is declared as EXTERNAL may have its body defined later on in the same module. Such a routine becomes an entry point.
- The CPAGE percent (%) statement conditionally does a page eject if less than a specified number of lines remain on the current listing page.
- The MAXLENGTH function returns the maximum length that a string variable can assume.
- The %CHECK TRUNCATE option enables (or disables) the checking for truncation of strings.
- The PASCALVS exec for invoking the compiler under CMS has been modified so that the specification of the operands allows greater flexability.
- New compiler options have been added, namely: LINECOUNT, PXREF, PAGEWIDTH, and LANGLVL.
- The catalogued procedures for invoking Pascal/VS in OS Batch have been simplified.
- The format of the output listing has been modified so that longer source lines may be accomodated.

- Multiple debugger commands may be entered on a single line by using a semicolon (;) as a separator.
- The format of the Pascal File Control Block has been modified.
- Support is now provided for ANSI and machine control characters on output files.
- Execution of a Pascal/VS program will terminate after a user determined number of non-fatal run time errors.
- The debugger now supports breakpoints at the end of a procedure or function.
- The Trace mode in the debugger provides information on when procedures are being exited.
- The TRACE procedure now permits you to specify the file on which the traceback is to be written.
- The Equate command of the debugger has been enhanced.
- The debugger will print "uninitialized" when displaying a variable that has not been assigned.
- New run time options are provided: SETMEM, ERRCOUNT, and ERRFILE.

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 (VS1 and 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 and later.

1.1 INVOKING THE COMPILER UNDER CMS: PASCALVS EXEC

PASCALVS	**fn [ft [fm]] [([options] [PRINT / NOPRINT / DISK] [LIB(maclibs)] [CONSOLE] [NOOBJ] [)]]**

fn is the file name of the source program.

ft is the file type of the source program; the assumed file type is **"PASCAL."**

fm is the file mode of the source program.

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

options are compiler options.

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

NOPRINT specifies that the listing is to be suppressed.

DISK specifies that the listing is to be stored as a file named **"fn LISTING."** This is the default.

CONSOLE specifies that the console messages produced by the compiler are be stored as a file named **"fn CONSOLE."** If **CONSOLE** is not specified, then the messages will be displayed on the terminal console.

NOOBJ suppresses the production of an object module.

1.2 BUILDING A LOAD MODULE UNDER CMS: PASCMOD EXEC

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

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.

options... is a list of 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 following are recognized as options to the PASCMOD command.

DEBUG links the debugging routines into the load module so that the interactive debugger can be used.

NAME name 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**."

1.3 INVOKING THE LOAD MODULE UNDER CMS

A Pascal/VS load module is invoked as follows:

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

where "**modname**" is the name of the load module; "**rtparms**" are run time options (separated by blanks); and "**parms**" are the parameters (if any) being passed.

1.4 INVOKING THE COMPILER UNDER TSO: PASCALVS CLIST

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]

data-set-name is the name of the primary input data set.

compiler-options-list is one or more compiler options separated by blanks

OBJECT(dsname) specifies the data set to contain the object module.

NOOBJECT specifies that no object module is to be produced.

PRINT(*) specifies that the compiler listing is to be displayed on the terminal.

PRINT(dsname) specifies the data set to contain the compiler listing.

SYSPRINT(sysout-class) specifies the sysout class to where the compiler listing is to be produced.

NOPRINT suppresses the compiler listing.

CONSOLE(*) specifies that compiler messages are to be displayed on the terminal.

CONSOLE(dsname) specifies the data set to contain compiler messages.

LIB('dsname-list') specifies a list of **%INCLUDE** libraries.

NOLIB specifies that no **%INCLUDE** libraries are required.

1.5 BUILDING A LOAD MODULE UNDER TSO: PASCMOD CLIST

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

data-set-name is the data set containing a Pascal/VS object module and/or linkage editor control cards.

OBJECT('dsname-list') specifies a list of data sets which contain additional object modules to be included in the link-edit.

LIB('dsname-list') specifies a list of libraries to be searched.

DEBUG specifies that the Pascal/VS interactive debugger is to be utilized.

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.

1.6 INVOKING THE LOAD MODULE UNDER TSO: THE CALL COMMAND

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

dsname(member) specifies the name of a partitioned data set and the member where the load module to be invoked is stored.

options is one or more run time options separated by either a comma or a blank.

parms a parameter string which is to be passed to the program.

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

1.7 INTERACTIVE DEBUGGER

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

- Compile the module to be debugged with the DEBUG option.
- When link-editing your program, include the debug library.
- When executing the load module, specify 'DEBUG' as a run time option.

Command name	Description (Abbreviation in capital letters)
?	List all debug commands
,variable	Display the value of a variable
Break	Set a break point
CLEAR	Remove all break points
Cms	Enter CMS subset mode
Display	Display status
Display Breaks	Display the location of all break points
Display Equates	Display all equate symbols with their current definitions
END	Terminate the program (same as QUIT)
Equate	Define an equate symbol
Go	Begin or resume execution of probram
Listvars	List the values of all variables that are local to the active routine
Qual	Redefine the "current" qualification
QUIT	Terminate the program (same as END)
Reset	Remove a break point
Set Attr	Display attributes when variables are viewed
Set Count	Initiate/terminate statement counting
Set Trace	Activate/deactive program tracing
Trace	Display a trace back
Walk	Execute a single statement and then prompt for another command

1.8 COMPILER OPTIONS

Compiler Option	Abbreviated Name	Default
CHECK/NOCHECK	---	CHECK
DEBUG/NODEBUG	---	NODEBUG
GOSTMT/NOGOSTMT	GS/NOGS	GOSTMT
LANGLVL(STANDARD/	STD	LANGLVL(EXTENDED)
STDRES/	---	
EXTENDED)	EXT	
LINECOUNT(n)	LC(n)	LINECOUNT(60)
LIST/NOLIST	---	NOLIST
MARGINS(m,n)	MAR(m,n)	MARGINS(1,72)
OPTIMIZE/NOOPTIMIZE	OPT/NOOPT	OPTIMIZE
PAGEWIDTH(n)	PW(n)	PAGEWIDTH(128)
PXREF/NOPXREF	---	PXREF
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)

1.9 RUN TIME OPTIONS

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

COUNT generates a statement count table and writes it to OUTPUT.

DEBUG activates the interactive debugger.

SETMEM initializes local storage of a routine to a specific value on each invocation of the routine.

NOSPIE suppresses the interception of program exceptions.

NOCHECK causes all checking errors to be ignored.

ERRFILE = ddname specifies the file to which error diagnostics are to be written.

ERRCOUNT = number specifies the number of non-fatal run time errors that will be permitted prior to terminating the program. The default number is 20.

MAINT Includes system run time routines in any error trace backs.

STACK = number specifies the number of kilobytes by which the run time stack is to be extended when a stack overflow occurs.

HEAP = number specifies the number of kilobytes by which the heap is to be extended when a heap overflow occurs.

1.10 CATALOGED PROCEDURES

PASCC Compile only -- step name: PASC

PASCCG Compile, load and execute -- step names: PASC, GO

PASCCL Compile and link-edit -- step name: PASC, LKED

PASCCLG Compile, link-edit, and execute -- step names: PASC, LKED, GO

Data set description	stepname.ddname
source program input	PASC.SYSIN[1]
%INCLUDE library (PDS)	PASC.SYSLIB
source listing, cross-reference listing, pseudo assembly listing and external symbol table listing	PASC.SYSPRINT
object module	PASC.SYSLIN
load module	LKED.SYSLMOD
linkage-editor control cards	LKED.SYSIN[1]
linkage-editor load library	LKED.SYSLIB
loader input	GO.SYSLIN
loader library	GO.SYSLIB
file OUTPUT	GO.OUTPUT

[1] This DDname is not defaulted and must be explicitly defined.

1.11 SAMPLE BATCH JOB

```
//jobname JOB
//STEP1   EXEC PASCCLG,OPTIONS='XREF(LONG),LIST'
//PASC.SYSIN DD *

  {Program to be compiled goes here}

/*
//LKED.SYSIN DD *
  ENTRY PASCALVS
/*
//GO.INPUT DD...
```

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 "Appendix A. Command Syntax Notation" on page 167.

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.

2.1 HOW TO COMPILE A PROGRAM

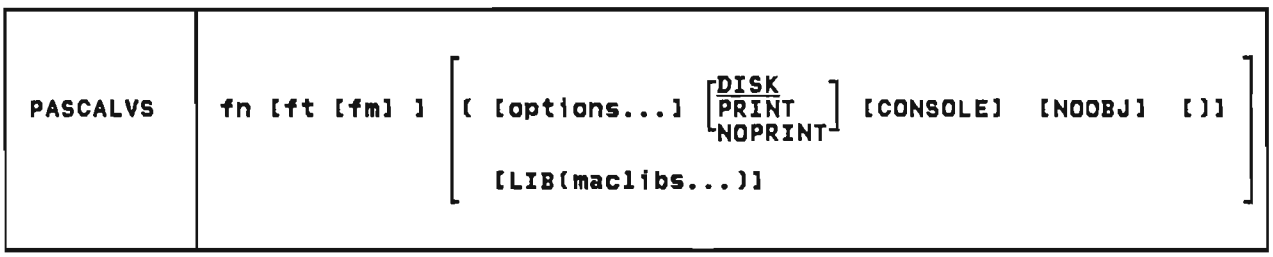


Figure 1. The PASCALVS command of CMS: invokes the Pascal/VS compiler.

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. If the file type is not explicitly specified, the type **"PASCAL"** will be assumed.

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 ft fm
is the file name, file type, and file mode of the source program. The file type and file mode are optional. The default file type is **"PASCAL"** and the default file mode is **"*."**

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 31.

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

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. This option is the default.

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

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

- NOSOURCE
- NOXREF
- NOLIST

CONSOLE
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 your terminal console.

NOOBJ
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 163.

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 31) 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 37.

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 133.

2.1.7 Sample Compilation

```
edit copy pascal
 NEW FILE:
program copy;
var
  infile,
  outfile  : text;
  buffer   : string(1000);
begin
  reset(infile);
  rewrite(outfile);
  while not eof(infile) do
    begin
      readln(infile,buffer);
      writeln(outfile buffer)
    end;
end.

 EDIT:

file
FILE SAVED

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

pascalvs copy

INVOKING PASCAL/VS R2.0

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

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

RETURN CODE: 8
R(00008); T=0.34/0.67 06:56:59
```

Figure 2. Sample compilation under CMS

2.2 HOW TO BUILD A LOAD MODULE

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

Figure 3. The PASCMOD command: generates a Pascal/VS 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
: specifies that the debugging routines are to be linked into the load module so that the interactive debugger can be used. (See "Pascal/VS Interactive Debugger" on page 67.)

NAME name
: 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 "Pascal/VS Interactive Debugger" on page 67 for a description of Debug).

2.3 HOW TO DEFINE FILES

```
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

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;
```

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. **Note**: A file that is being defined to the terminal requires you to explicitly specify RECFM and LRECL on the FILEDEF command.

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 45.

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...]
```

For a description of the run time options see "Run Time Options" on page 35.

This section describes how to compile and execute 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 "Appendix A. Command Syntax Notation" on page 167 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.

3.1 HOW TO COMPILE A PROGRAM

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

3.1.1 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 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 31.

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 qualifier 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, then the name of the associated object module will be generated as just described. If the object module data set is a partitioned data set, then the object module will become a member within the PDS and will have the same name as the member name of the input 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(MAIN)'

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

PRINT(*)
specifies that the compiler listing is to be displayed on 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 17.

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

---

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

Example 1

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)
```

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.

---

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

For a synopsis of the compiler generated messages see "Pascal/VS Compiler Messages" on page 133.

3.2 HOW TO BUILD A LOAD MODULE

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

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). 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.

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

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 **PASCALVS**.

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 **highlighted**.)

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

READY
```

3.3 HOW TO DEFINE FILES

```
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

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;
```

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 45.

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.

3.4 INVOKING THE LOAD MODULE

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

Figure 8. The TSO CALL command to invoke a 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" on page 35.).

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.

3.5 SAMPLE TSO SESSION

```
READY

 pascalvs lander sysprint(a) list

INVOKING PASCAL/VS R2.1
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

```
//EXAMPLE JOB
//STEP1   EXEC PASCCG,PARM='LIST'
//PASC.SYSIN DD *
 program EXAMPLE(INPUT,OUTPUT);
 var
   A, B: REAL;
 begin
   RESET(INPUT);
   while not EOF(INPUT) 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

The job control statements shown in Figure 10 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 29 and "Using Input/Output Facilities" on page 45.

Any options to be passed to the compiler are placed within the PARM string of the EXEC statement.

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 two steps:

PASC invokes the Pascal/VS compiler to produce an object module.

GO invokes the LOADER 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 SYSLIN in procedure step PASC 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
//PASC.SYSLIN DD DSN=T123.SORT.OBJ,
//               UNIT=TSOPACK,
//               DISP=(NEW,CATLG)
```

4.4.1 Compile Only (PASCC)

```
//PASCC   PROC SYSOUT='*',INCLLIB='SYS1.PASCALVS.MACLIB'
//*
//*      INVOKE PASCAL/VS COMPILER
//*
//PASC    EXEC  PGM=PASCALI,PARM=,REGION=512K
//OUCODE    DD  SYSOUT=&SYSOUT
//OUTPUT    DD  SYSOUT=&SYSOUT
//STEPLIB   DD  DSN=SYS1.PASCALVS.LINKLIB,DISP=SHR
//SYSLIB    DD  DSN=&INCLLIB,DISP=SHR
//          DD  DSN=SYS1.PASCALVS.MACLIB,DISP=SHR
//SYSLIN    DD  DSNAME=&&LOADSET,UNIT=SYSDA,DISP=(MOD,PASS),
//              SPACE=(TRK,(2,5)),
//              DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DSORG=PS)
//SYSLIST   DD  UNIT=SYSDA,DISP=(NEW,DELETE),
//              SPACE=(TRK,(2,5))
//SYSMSGS   DD  DSN=SYS1.PASCALVS.MESSAGES,DISP=SHR
//SYSOIN    DD  UNIT=SYSDA,DISP=(NEW,DELETE),
//              SPACE=(TRK,(2,5))
//SYSPRINT  DD  SYSOUT=&SYSOUT,DCB=(RECFM=VBA,LRECL=133,BLKSIZE=685)
//SYSTERM   DD  DUMMY
//SYSTIN    DD  UNIT=SYSDA,DISP=(NEW,DELETE),
//              SPACE=(TRK,(2,5))
//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)
//SYSXREF   DD  UNIT=SYSDA,DISP=(NEW,DELETE),
//              SPACE=(TRK,(2,5))
//UCODE     DD  SYSOUT=&SYSOUT
```

Figure 11. Cataloged procedure PASCC

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

Step PASC reads in the source module, diagnoses errors, produces a listing, and generates an object module to the data set associated with DDname SYSLIN.

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.

4.4.2 Compile, Load, and Execute (PASCCG)

```
//PASCCG  PROC SYSOUT=*,INCLLIB='SYS1.PASCALVS.MACLIB',
//             LKLBDSN='SYS1.PASCALVS.LOAD',
//             LINKLIB='SYS1.PASCALVS.LINKLIB'
//PASC    EXEC  PGM=PASCALI,PARM=,REGION=512K

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

//GO      EXEC  PGM=LOADER,COND=(8,LE,PASC),PARM='EP=PASCALVS'
//OUTPUT    DD  SYSOUT=&SYSOUT,DCB=(RECFM=VBA,LRECL=133,BLKSIZE=685)
//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,DCB=(RECFM=VBA,LRECL=133)
```

Figure 12. Cataloged procedure 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
```

4.4.3 Compile and Link Edit (PASCCL)

```
//PASCCL   PROC SYSOUT=*,INCLLIB='SYS1.PASCALVS.MACLIB',
//              LKLBDSN='SYS1.PASCALVS.LOAD',
//              LINKLIB='SYS1.PASCALVS.LINKLIB'
//PASC     EXEC  PGM=PASCALI,PARM=,REGION=512K

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

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

Figure 13. Cataloged procedure 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 PASCALVS. 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 PASCALVS
/*
```

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 28 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 PASCALVS
/*
```

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

4.4.4 Compile, Link Edit, and Execute (PASCCLG)

```
//PASCCLG PROC SYSOUT=*,INCLLIB='SYS1.PASCALVS.MACLIB',
//             LKLBDSN='SYS1.PASCALVS.LOAD',
//             LINKLIB='SYS1.PASCALVS.LINKLIB'
//PASC     EXEC  PGM=PASCALI,PARM=,REGION=512K

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

//LKED     EXEC  PGM=IEWL,PARM='LIST,MAP',COND=(8,LE,PASC)

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

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

Figure 15. Cataloged procedure 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 two steps of this procedure are identical to those of the PASCCL procedure. An additional third 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 file named OUTPUT 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 a DD statement for OUTPUT. OUTPUT is assigned to the output class where the system messages and compiler listings are produced (SYSOUT=*).

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 45. 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,PARM='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,PARM='NOXREF'
//PASC.SYSIN DD *
segment IO;
type
  F80    = file of
             packed array[1..80] of CHAR;
procedure COPY(var FIN,FOUT: F80);
  external;

procedure COPY;
begin
  while not EOF(FIN) do
    begin
      FOUT@ := FIN@;
      PUT(FOUT);
      GET(FIN)
    end
end;.
/*
//LKED.SYSIN DD *
  ENTRY PASCALVS
/*
//GO.INFILE DD *
          ...
  (data to be copied into data set goes here)
          ...
/*
//GO.OUTFILE DD DSN=P123456.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

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
LANGLVL(STANDARD)/ LANGLVL(STDRES)/ LANGLVL(EXTENDED)	LANGLVL(STD)/ ---/ LANGLVL(EXT)	LANGLVL(EXTENDED)
LINECOUNT(n)	LC(n)	LINECOUNT(60)
LIST/NOLIST	---	NOLIST
MARGINS(m,n)	MAR(m,n)	MARGINS(1,72)
OPTIMIZE/NOOPTIMIZE	OPT/NOOPT	OPTIMIZE
PAGEWIDTH(n)	PW(n)	PAGEWIDTH(128)
PXREF/NOPXREF	---	PXREF
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; however, to help you detect such errors, the SETMEM runtime option has been provided (see "Run Time Options" on page 35).

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.

String truncation
Assignments to varying length strings are checked to make sure that the destination string variable is declared large enough to contain the source string.

When a runtime checking error occurs, a diagnostic message will be displayed on your terminal followed by a traceback of the routines which were active when the error occurred. If the program is invoked from OS Batch, the diagnostic message and traceback will be sent to the data set or device associated with DDname SYSPRINT. You may direct the error diagnostics to any file of your choice with the "ERRFILE" option (see "Run Time Options" on page 35).

See "Reading a Pascal/VS Trace Back" on page 61 for an example of a traceback due to a checking error.

"User Handling of Execution Errors" on page 64 describes how checking errors may be intercepted by your program.

5.2 DEBUG/NODEBUG

An interactive debugging facility is available to debug Pascal/VS programs. The debugger is described in "Pascal/VS Interactive Debugger" on page 67. 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 "Pascal/VS Interactive Debugger" on page 67.

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 LANGLVL()

If LANGLVL(STANDARD) is specified, the compiler will diagnose all constructs and features which do not conform to "standard" Pascal. Violations of the standard will appear as warnings. In addition, many of the predeclared identifiers which are unique to Pascal/VS will not be recognized when LANGLVL(STANDARD) is specified.

If LANGLVL(STDRES) is specified, the compiler will turn LANGLVL(STANDARD) on, and will also not recognize any of the non-ANSI-standard Pascal/VS reserved words. This means that the following Pascal/VS reserved words may now be used as identifiers (of course, the features they support are lost, too):

assert
continue
def
leave
otherwise
range
ref
return
space
static
value
xor

LANGLVL(EXTENDED), which is the default, specifies that the full Pascal/VS language is to be supported.

5.5 LINECOUNT(N)

The LINECOUNT option specifies the number of lines to appear on each page of the output listing. The maximum number of lines to fit on a page depends on the form to which the output is being printed.

The default is 60 lines to the page.

5.6 LIST/NOLIST

The LIST/NOLIST option controls the generation or suppression of the translator pseudo-assembler listing (see "Assembly Listing" on page 42).

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

5.7 MARGINS(M,N)

MARGINS('1,72')

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 100 The specified margins must not overlap the sequence field.

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,

5.8 OPTIMIZE/NOOPTIMIZE

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

When code is being optimized, the code generation phase of the compiler will try to eliminate common subexpressions. Instead of evaluating an expression each time it occurs in the program, the expression will be evaluated once and saved, if possible, in a register. The example in Figure 17 illustrates this.

Sample program to demonstrate code optimization

```
program TEST;
var
  I,J,K : integer;
begin
  I := 80;
  J := I * 3;
  J := 2;
  K := I * 3;
  K := 2;
end.
```

Optimized code

```
*  I := 80;
          LA     03,80
          ST     03,144(,13)
*  J := I * 3;
          MH     03,=H'3'
          ST     03,148(,13)
*  J := 2;
          LA     04,2
          ST     04,148(,13)
*  K := I * 3;
          ST     03,152(,13)
*  K := 2;
          ST     04,152(,13)
```

Unoptimized code

```
*  I := 80;
          LA     03,80
          ST     03,144(,13)
*  J := I * 3;
          L      03,144(,13)
          MH     03,=H'3'
          ST     03,148(,13)
*  J := 2;
          LA     03,2
          ST     03,148(,13)
*  K := I * 3;
          L      03,144(,13)
          MH     03,=H'3'
          ST     03,152(,13)
*  K := 2;
          LA     03,2
          ST     03,152(,13)
```

Figure 17. Differences between OPT and NOOPT

5.9 PAGEWIDTH(N)

The PAGEWIDTH option specifies the maximum number of characters[5] that may appear on a single line of the output listing. This number depends on the page form and the printer model.

The default page width is 128 characters, with the minimum and maximum page widths allowed being 120 and 210 characters, respectively.

[5] The number specified in the PAGEWIDTH option does not include carriage control characters.

5.10 PXREF/NOPXREF

The PXREF option specifies that the right margin of the output listing is to contain cross reference entries (see "Page Cross Reference Field" on page 38). NOPXREF suppresses these entries.

5.11 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; they 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')
```

5.12 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.

5.13 WARNING/NOWARNING

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

5.14 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 40).

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.

Features within the Pascal/VS run time environment may be enable or disabled by passing options to the Pascal/VS program. These options are passed to a Pascal/VS program through the parameter passing mechanism. To distinguish run time options from the parameter string intended to be processed by the program, the options must preceed the parameter string (if any) and be terminated with a slash ("/").

The following is a list of supported run time options.

COUNT
specifies that instruction frequency information is to be collected during program execution. After the program is completed, this information is written to file OUTPUT.

Note: This option will only have an effect if the program was both compiled and link-edited with the DEBUG option.

DEBUG
specifies that the interactive debugger (see "Pascal/VS Interactive Debugger" on page 67) is 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 (see "Module Generation Options" on page 12).

ERRCOUNT=n

ERRCOUNT(n)
specifies how many non-fatal errors are allowed to occur before the program is abnormally terminated. The default is 20.

Note to CMS users: due to the 8-character tokenization convention of CMS, a blank must precede the '=' symbol in the ERRCOUNT specification.

Example:

```
modname ERRCOUNT =1/
```

ERRFILE=ddname

ERRFILE(ddname)
specifies the DDname of the file to which all run time diagnostics are to be written. Under CMS and TSO, diagnostics are displayed on your terminal by default. Under OS batch, the default error file is SYSPRINT.

Note to CMS users: due to the 8-character tokenization convention of CMS, the '=' symbol must be surrounded with blanks.

Example:

```
modname ERRFILE = OUTPUT/
```

HEAP = n
specifies the number of kilobytes[6] that the heap is to be "extended" each time the heap overflows. The heap is where memory is allocated when the procedure NEW is called. When the end of the heap is reached, the GETMAIN supervisor call is invoked to allocate more memory for the heap. If the length of the space being required by NEW is greater than "n," then the amount to be allocated will be the length of the space rounded up to the next kilobyte (1024 bytes).

There is a significant overhead penalty for each invocation of GETMAIN. If "n" is too small, GETMAIN will be invoked frequently and the execution speed of the program will be affected. If "n" is too large, the heap will contain memory that is never used.

The default HEAP attribute is 12 kilobytes.

MAINT
specifies that when a run time error occurs, the trace back is to list active run time support routines. These routines begin with an AMP prefix and are normally suppressed from the trace back listing. This option is used to locate bugs within the run time environment.

NOCHECK
specifies that any checking errors detected within the program are to be ignored.

NOSPIE
specifies that the Pascal/VS run time enviroment is not to issue a SPIE request and therefore will not intercept program interrupts.

STACK = n
specifies the number of kilobytes[6] that the run time stack is to be "extended" each time the stack overflows. The run time stack is where the dynamic storage area (DSA) of a routine is allocated when the routine is invoked. When

[6] A "kilobyte" is defined as 1024 bytes in the context of this manual.

the end of the stack is reached, the GETMAIN supervisor call is invoked to allocate more memory for the stack. If the length of the DSA being required is greater than "n," then the amount to be allocated will be the length of the DSA rounded up to the next kilobyte (1024 bytes).

There is a significant overhead penalty for each invocation of GETMAIN. If "n" is too small, GETMAIN will be invoked frequently and the execution speed of the program will be affected. If "n" is too large, the stack will occupy more memory than is necessary.

The default STACK attribute is 12 kilobytes.

SETMEM
specifies that upon entry to each Pascal/VS routine, each byte of memory in which the routine's local variables are allocated will be set to a specific value, namely 'FE' (hexadecimal). This option aids in locating the source of intermittent errors which occur because of the use of uninitialized variables.

7.1 SOURCE LISTINGS

```
PASCAL/VS RELEASE 2.0     UTILITY:      01/27/81  14:48:54        PAGE 5

 B P C I  STMT #                    SOURCE PROGRAM                PAGE XREF
                        INCLUDE 1 FROM SYSLIB  (GLOBALS )
                    V---+----1----+----2----+----3---//--7-V SEQ NO
           1:       |                                              00000100
           1:       |type                                          00000200 R
           1:       |  NAMEPTR = @NAMEREC;                         00000300 * *
           1:       |  NAMEREC =                                   00000400 *
           1:       |       record                                 00000500 R
           1:       |         NAME      : STRING(30);              00000600 * P
           1:       |         LEFT_LINK,                           00000700 *
           1:       |         RIGHT_LINK: NAMEPTR;                 00000800 * 5
           1:       |       end;                                   00000900 R
           1:       |                                              00001000
           1:       |def                                           00001100 R
           1:       |  TREETOP : NAMEPTR;                          00001200 * 5
                    |                                              00000180
   1                |procedure SEARCH(                             00000190 R *
   1                |      const  ID: STRING;                      00000200 R * P
   1                |        var PTR: NAMEPTR);                    00000210 R * 5
   1                |EXTERNAL;                                     00000220 *
                    |                                              00000221
   1                |procedure SEARCH;                             00000222 R *
   1                |var                                           00000230 R
   1                |  LPTR  = NAMEPTR;                            00000240 * 5
==========ERROR=>          $17
   1                |begin                                         00000250 R
   1           1    |  PTR := nil;                                 00000260 5 P
   1           2    |  LPTR := TREETOP;                            00000270 5 5
   1    1      3    |  while LPTR <> nil do                        00000280 R 5 P R
   1    1           |    begin                                     00000290 R
 1 1    1      4    |      with LPTR@ do                           00000300 R 5 R
 1 1 1 1       5    |        if NAME = ID then                     00000310 R 5 5 R
 1 1 1 1            |          begin                               00000320 R
 2 1 1 1       6    |            PTR := LPTR                       00000330 5 5
 2 1 1 1       7    |            return                            00000340 R
==========ERROR=>                $8
 1 1 1 1            |          end                                 00000350 R
 1 1 1 1            |        else                                  00000360 R
 1 1 2 1       8    |          if ID < NAME then                   00000370 R 5 5 R
 1 1 2 1       9    |            LPTR := LEFT_LINK                 00000380 5 5
 1 1 2 1            |          else                                00000390 R
 1 1 2 1      10    |            LPTR := RIGHT_LINK                00000400 5 5
   1    1           |    end (*while*)                             00000410 R
                    |end;.                                         00000420 R

 NUMBER OF ERRORS DETECTED:   2

 DIAGNOSTIC MESSAGES ON PAGE(S):    5

 ERROR     8: SEMICOLON ";" EXPECTED
 ERROR    17: ":" EXPECTED

 PARAMETERS PASSED: DISK NOXREF LIB ( MACLIB )

 OPTIONS IN EFFECT: MARGINS(1,72), SEQ(73,80), LINECOUNT(60), CHECK,
                    GOSTMT, OPTIMIZE, PXREF, SOURCE, WARNING

 SOURCE LINES:  53;   COMPILE TIME: 0.43 SECONDS;  COMPILE RATE: 7441 LPM
```

Figure 18. Sample source listing

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

7.1.1 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.

7.1.2 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:

```
B P C I    STMT
```

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

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

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.

7.1.3 Statement Numbering

Pascal/VS numbers the statements of a routine. These numbers are referenced when a run time error occurs (see "Reading a Pascal/VS Trace Back" on page 61) and when break points are specified in the interactive debugger (see "Pascal/VS Interactive Debugger" on page 67).

All non-empty statements are numbered except the repeat statement. However, the **until** portion of a repeat statement is numbered.

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.

7.1.4 Page Cross Reference Field

If the PXREF compiler option is active, the right margin of the listing contains a cross reference field. This field 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.

7.1.5 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.

7.1.6 Option List

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

7.1.7 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.

```
                    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 #>

ID               5/20       IN SEARCH, CLASS = CONST PARAMETER,
                            TYPE = STRING, OFFSET = 144
                             5/31          5/37

LEFT_LINK        5/1:7      IN NAMEREC, CLASS = FIELD, TYPE = POINTER,
                            OFFSET = 32, LENGTH = 4
                             5/38

LPTR             5/24       IN SEARCH, CLASS = LOCAL VAR, TYPE = POINTER,
                            OFFSET = 152, LENGTH = 4
                             5/27          5/28          5/30          5/33
                             5/38          5/40

NAME             5/1:6      IN NAMEREC, CLASS = FIELD, TYPE = STRING,
                            OFFSET = 0, LENGTH = 32
                             5/31          5/37

NAMEPTR          5/1:3      CLASS = TYPE, TYPE = POINTER, LENGTH = 4
                             5/1:8         5/1:12        5/21          5/24

NAMEREC          5/1:4      CLASS = TYPE, TYPE = RECORD, LENGTH = 40
                             5/1:3

NIL            PREDEFINED   CLASS = CONSTANT, TYPE = POINTER
                             5/26          5/28

PTR              5/21       IN SEARCH, CLASS = VAR PARAM, TYPE = POINTER,
                            OFFSET = 148, LENGTH = 4
                             5/26          5/33

RIGHT_LINK       5/1:8      IN NAMEREC, CLASS = FIELD, TYPE = POINTER,
                            OFFSET = 36, LENGTH = 4
                             5/40

SEARCH           5/19       CLASS = ENTRY PROCEDURE

STRING         PREDEFINED   CLASS = TYPE, TYPE = STRING
                             5/1:6         5/20

TREETOP          5/1:12     CLASS = DEF VAR, TYPE = POINTER, LENGTH = 4
                             5/27
```

Figure 19. 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 meanings.

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 external entry point.

ENTRY PROCEDURE procedure routine declared as an external 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

MAIN ENTRY POINT procedure declared as MAIN whose body is not in this module

PROCEDURE a user-defined or standard procedure

REENTRANT ENTRY POINT procedure declared as REENTRANT whose body is not in this module

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.

```
PASCAL/VS RELEASE 2.0     UTILITY :          01/27/81  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 20. 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.

7.4 EXTERNAL SYMBOL DICTIONARY

```
PASCAL/VS RELEASE 2.0      AMPLXREF:          01/27/80  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 21. Sample ESD table

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.

7.5 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.

8.1 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). The Basic Direct Access Method (BDAM) is used for random record access.

8.2 DDNAME ASSOCIATION

For any identifier declared as a simple 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.

An explicit DDname may be associated with a file variable by means of the DDNAME option when the file is opened. (see "The Open Options" on page 56).

DDnames should be explicitly specified for files which are elements of arrays, fields of records, or pointer qualified. If the DDname is not explicitly specified for such files, a DDname of the form "PASCALnn" will be assigned to the file, where "nn" is a two digit integer.

8.3 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 all of the record formats that are supported by QSAM, such as, F, V, U, FB, VB, FBA, VBM, etc.

A Pascal/VS program will process a file that contains ANSI or machine control characters at the beginning of each logical record (in which case the record format would be specified as RECFM=...A or RECFM=...M). Each logical record written to such files will be prefixed with the appropriate control character. Thus, the first character position of each record is not directly accessible from the Pascal/VS program. (If the NOCC option is specified when the file is opened, no control character will be prefixed and the first character <u>is</u> accessible. See "The Open Options" on page 56.)

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 when the file is open by means of the options string. (see "The Open Options" on page 56).

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.

8.4 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 EBCDIC.

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.

8.5 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.

8.6 OPENING A FILE FOR INPUT - RESET

To explicitly open a file for input, the procedure RESET is used. A call to RESET has the forms:

```
RESET(f)
  or
RESET(f,options)
```

where "f" is a file variable and "options" is a string which contains the open options (see "The Open Options" on page 56). The "options" parameter may be omitted.

Normally, RESET allocates a buffer, reads in 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);
  (*get first character of file*)
  C := SYSIN@;
end.
```

Figure 22. Using RESET on a text file

8.7 OPENING A FILE FOR INTERACTIVE INPUT

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. For example, if the file being opened is assigned to your terminal, you will be prompted for data when the file is opened. This may not be preferable if your program is suppose to write out prompting messages prior to reading.

To alleviate this problem, a file may be opened for interactive input by specifying "INTERACTIVE" in the options string of RESET. 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). The end-of-line condition (see "End of Line Condition" on page 53) is initially set to TRUE.

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

Figure 23. Opening a file for interactive input

8.8 OPENING A FILE FOR OUTPUT - REWRITE

The procedure REWRITE is used to open a file for output. A call to the procedure has the forms:

```
REWRITE(f)
  or
REWRITE(f,options)
```

where "f" is a file variable and "options" is a string which contains the open options (see "The Open Options" on page 56). The "options" parameter may be omitted.

REWRITE positions the file pointer 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 24. Opening a text file with REWRITE

```
program EXAMPLE;
var
  OUTFILE : file of INTEGER;
  I       : INTEGER;
begin
  REWRITE(OUTFILE,
  'BLKSIZE=1600,LRECL=4,RECFM=F');
    ...
  OUTFILE@ := I;
  PUT(OUTFILE);
end.
```

Figure 25. Opening a record file with REWRITE

8.9 TERMINAL INPUT/OUTPUT

Two procedures are provided for doing input and output directly to your terminal without going through the normal DDname interface. Calls to these procedures have the forms:

```
TERMIN(f) or TERMIN(f,options)
TERMOUT(f) or TERMOUT(f,options)
```

where "f" is a text file variable and "options" is a string which contains the open options (see "The Open Options" on page 56). The "options" parameter may be omitted.

The TERMIN procedure opens a text file for interactive input from your terminal. Likewise, the TERMOUT procedure opens a text file for terminal output.

There is no concept of an end-of-file condition for files opened with TERMIN. The EOF function always returns FALSE for such files.

Note: The TERMIN procedure opens the file with the INTERACTIVE attribute as was described in "Opening a File for Interactive Input" on page 46.

```
program EXAMPLE;
var
  TTYIN, TTYOUT: text;
  I      : INTEGER;
begin
  TERMIN(TTYIN); TERMOUT(TTYOUT);
  WRITELN(TTYOUT,'ENTER DATA:');
  READLN(TTYIN,I);
    ...
end.
```

Figure 26. Terminal input/output example

8.10 OPENING A FILE FOR UPDATE

The UPDATE procedure is provided for opening a record file for updating. In this mode, records may be read, modified, and then replaced. A call to the procedure has the forms:

```
UPDATE(f)
  or
UPDATE(f,options)
```

where "f" is a record file variable and "options" is a string which contains the open options (see "The Open Options" on page 56). The "options" parameter may be omitted.

Upon calling UPDATE, a file buffer is allocated and the first record of the

file is read into it. If a subsequent PUT operation is performed on the file, the contents of the buffer will be stored back into the file at the location from which it was read.

Each GET operation reads in the next subsequent record of the file. A PUT operation will write the record back from where the last GET operation obtained it.

```
program EXAMPLE;
var
  F    : file of
           record
              NAME: STRING(30);
              AGE : 0..99;
           end;
begin
  UPDATE(F);
  (*update each record     *)
  (*  by incrementing age *)
  while not EOF(F) do
    begin
      F@.AGE := F@.AGE + 1;
      PUT(F);
      GET(F)
    end;
end.
```

Figure 27. Updating a record file

8.11 PROCEDURE GET

The GET procedure is the means by which a basic read operation is performed on a file. A call to the procedure has the form:

```
GET(f)
```

where "f" is a file variable.

8.11.1 GET operation on text files

When applied to an input text file, GET 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" on page 53) 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, prior to the call to GET, the file pointer is positioned to the end of the last logical record of a text file (in which case the end-of-line condition will be true) then the end-of-file condition will become true. (See "End of File Condition - text files" on page 54).

If GET is attempted on a text file that has not been opened, it will be implicitly opened for input (as if RESET had been called).

```
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 28. Using GET on a text file

8.11.2 GET operation on record files

Each call to GET for the case of record files reads the next sequential logical record into the buffer referenced by the file pointer. The end-of-file condition will become true if there are no more records within the file, in which case, the file pointer will be set to **nil**.

A record file must be opened for input or update prior to executing a GET operation, otherwise, a runtime diagnostic will be generated.

```
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 29. Using GET on record files

8.12 PUT PROCEDURE

The PUT procedure is the means by which a basic write operation is performed on a file. A call to the procedure has the form:

```
PUT(f)
```

where "f" is a file variable.

The file must be opened for output or update prior to calling PUT[7]; otherwise, a runtime diagnostic will occur.

8.12.1 PUT Operation on Text Files

The PUT procedure, when applied to a text file opened for output, 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 53).

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

Figure 30. Using PUT on a text file

8.12.2 PUT Operation on Record Files

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 31. Using PUT on record files

8.13 TEXT FILE PROCESSING

8.13.1 Text File READ

The READ procedure fetches data from a text file beginning at the current position of the file pointer. A call to the procedure has the forms:

```
READ(f,v)
  or
READ(f,v:n)
```

where "f" is a file variable and "v" is a variable which must be of one of the following types:

CHAR (or a subrange thereof)

INTEGER (or a subrange thereof)

packed array[] **of** CHAR

REAL (or SHORTREAL)

STRING

"n" is an optional field length (an integer expression). The file variable

[7] 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.)

"f" may be omitted, in which case, the file INPUT is assumed.

A call of the form

```
READ(f,v1,v2,...vn)
```

is executed as

```
begin
  READ(f,v1);
  READ(f,v2);
    ...
  READ(f,vn);
end
```

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

Upon executing READ, if the file pointer is not yet set, an initial GET operation is performed. This case occurs when a file is opened **INTERACTIVE**ly. (see "Opening a File for Interactive Input" on page 46.)

When reading INTEGER or REAL data via the READ procedure, and no field length 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, an error diagnostic will be produced.

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:

```
951235AN 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.

If a field length 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     : 0..99999;
  MAN     : 0..999999;
  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 32. 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 field length (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 33. Using READ on text files

8.13.2 The READLN Procedure

A call to READLN has the same form as a call to READ and performs the same function except that after the data has been read, all remaining characters within the logical record are skipped. The procedure is applicable to text files only.

Normally, READLN causes the next logical record to be read (unless the end-of-file is reached) and the file pointer is positioned to the beginning of the buffer that contains the record.

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.

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 as though RESET had been called.

```
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 34. Using the procedure READLN

8.13.3 Text File WRITE

The WRITE procedure writes data to a text file beginning at the current position of the file pointer. A call to the procedure has the forms:

```
WRITE(f,e)
  or
WRITE(f,e:n)
  or
WRITE(f,e:n1:n2)
```

where "f" is a file variable and "e" is an expression which must be of one of the following types:

BOOLEAN

CHAR (or a subrange thereof)

INTEGER (or a subrange thereof)

packed array[] **of** CHAR

REAL (or SHORTREAL)

STRING

"n,""n1," and "n2" are optional field lengths (integer expressions). The file variable "f" may be omitted, in which case, the file OUTPUT is assumed.

A call of the form

```
WRITE(f,e1,e2,...en)
```

is executed as

```
begin
  WRITE(f,e1);
  WRITE(f,e2);
  ...
  WRITE(f,en);
end
```

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, a run time error diagnostic will be generated.

If a field length is specified for an expression to be written and its value is positive, the data will appear right justified in the output field. If the specified length is negative, the data will appear left justified. (The field width will be the absolute value of the specified length.)

String data that is being written with a specified field length will be truncated on the right if the field length is too small.

If no field length is specified, a default will be used that depends on the data's type:

data type	default field length
BOOLEAN	10
CHAR	1
INTEGER	12
REAL	20
SHORTREAL	20

In addition, expressions of type STRING have a default field length equal to their current length. Fixed length strings (**packed array of** CHAR) have a default equal to their declared length.

```
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:-30,
                      AGE:3,'  ');
        WRITE(OUTFILE,WEIGHT:3:0);
        WRITELN(OUTFILE)
      end;
end.
```

Figure 35. Using WRITE on text files

8.13.4 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 36. Using the WRITELN procedure

8.13.5 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). A call to the procedure has the following form:

```
PAGE(f)
```

where "f" is a variable of type TEXT which has been opened for output.

If a logical record is partially filled, an implicit WRITELN will be performed prior to the page eject.

For this procedure to produce any effect, the first character of each logical record of the file must be reserved for carriage control. This is done by specifying either A (ANSI control) or M (machine control) in the RECFM attribute for the file.

If the record format specifies ANSI control, then the character '1' will be inserted in the first character position of the record. For machine control, a single record is written that contains the hexadecimal value of '8B' in its first character position.

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

Figure 37. Using the PAGE procedure

8.13.6 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 38. Using the EOLN function

8.13.7 End of File Condition - text files

The end-of-file condition becomes true for a text file when one of the following occurs:

- RESET is called and the file is empty.
- The file is open for output.
- GET is called when the file pointer is positioned at the end of the last logical record of the file (in which case the end-of-line condition is true).
- READ is called and all characters of the last logical record were consumed.

When the end-of-file condition occurs, the file pointer has the value **nil**.

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 39. Using the EOF function on a text file

8.14 RECORD FILE PROCESSING

8.14.1 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, an error diagnostic will be generated at run time.

8.14.2 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, an error diagnostic will be generated at run time.

```
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 40. Using READ and WRITE on record files

8.14.3 End of File Condition - Record Files

The end-of-file condition becomes true for a record file when:

- RESET is called for an empty file.

- The file is opened for output.
- GET is executed for a file in which no more records remain.

When the end-of-file condition occurs, the file pointer has the value **nil**. 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 produce an error diagnostic.

8.15 CLOSING A FILE

The procedure CLOSE is provided to close a file explicitly. A call to this procedure has the form

```
CLOSE(f)
```

where "f" is a file variable.

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

If the variable associated with an open file is destroyed prior to program termination, the results could be disastrous when Pascal/VS attempts to close the file. This problem could occur in the following cases:

- the file variable is an element of an array.
- the file variable is a field of a record.
- the file variable is pointer qualified (exists on the heap).
- a routine which contains local file variables is exited with a **goto** statement.

In these cases, the file variable must be closed explicitly with a call to CLOSE.

```
program EXAMPLE;
type
var
  FSTK   : array[1..8] of
             TEXT;
  DDNAME: STRING(8);
  I      : 1..8;
begin
  ...
  RESET(FSTK[I],'DDNAME='||DDNAME);
  ...
  for I := 1 to 8 do
     CLOSE(FSTK[I]);
end.
```

Figure 41. Example of using CLOSE

8.16 RELATIVE RECORD ACCESS

Pascal/VS permits records of a record file to be accessed in a random order by means of the SEEK procedure. A call to SEEK has the form

```
SEEK(f,n)
```

where "f" is a record file that was previously opened with RESET, REWRITE, or UPDATE; "n" is a positive integer expression which corresponds to a record number. The the number of the first record is 1.

A subsequent call to GET or PUT will operate on the "nth" record of the file. Each call to GET or PUT thereafter will operate on subsequent records. SEEK does not perform an I/O operation.

At the first call to SEEK, the file is implicitly closed and reopened for random access using the Basic Direct Access Method (BDAM). The file that is to be accessed in this manner must have unblocked, fixed-length records; that is, the RECFM attribute for the file must be "F."

Under TSO and OS Batch, the first SEEK operation on a file opened with REWRITE will cause dummy records to be written to the associated data set until the file's primary space allocation is filled. The record number specified must not exceed the number of blocks in the file's primary space allocation.

Under CMS, the corresponding FILEDEF of a file being accessed with SEEK must have the XTENT attribute specified[8]. This attribute specifies the largest record number that may be accessed; however, it has nothing to do with the space occupied by the file. Thus, a FILEDEF specification of the form

[8] If the XTENT attribute is not specified, CMS will default it to 50.

```
FILEDEF F DISK FILE DATA(XTENT 65535
```

will permit any record in file F to be referenced with SEEK, regardless if it actually exists. If a record is being read that does not exist, CMS will return a buffer of zeroes.

```
program EXAMPLE;
type
  REC = record
          NAME : STRING(25);
          AGE  : 0..99;
          SEX  : (MALE,FEMALE)
        end;
  IDX = record
          RECNO: 0..MAXINT;
        end
var
  RECFILE: file of REC;
  IDXFILE: file of IDX;
begin
  RESET(IDXFILE);
  RESET(RECFILE);
  (*write out names in order of
    index                        *)
  while .not EOF(IDXFILE) do
    begin
      SEEK(RECFILE,IDXFILE@.RECNO);
      GET(RECFILE);
      WRITELN(OUTPUT,RECFILE@.NAME)
      GET(IDXFILE);
    end
end.
```

Figure 42. Example of using SEEK to access records randomly

8.17 PARTITIONED DATA SETS

8.17.1 Opening a Partitioned Data Set

To open a partitioned data set (PDS)[9], the procedures PDSIN and PDSOUT are provided. Calls to these procedures are of the form

```
PDSIN(f,options)
PDSOUT(f,options)
```

where "F" is a file variable and "options" is a string expression which contains open options (see "The Open Options"). Unlike the other procedures which open files, the options string is required and must specify a member name (MEMBER=name).

PDSIN opens the specified member in the PDS for input. As in the case of RESET, the file pointer is made to point to a buffer containing the first logical record of the file.

PDSOUT creates a member in the PDS and opens it for output. If the member already exists, it will be erased and then recreated.

See Figure 44 on page 58 for an example of opening a partitioned data set.

8.17.2 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
```

8.18 THE OPEN OPTIONS

All Pascal/VS procedures which open files are defined with an optional string parameter which contains options pertaining to the file being opened. 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:

[9] All operations that may be applied to "partition data sets" under OS may be applied to MACLIB's and TXTLIB's under CMS.

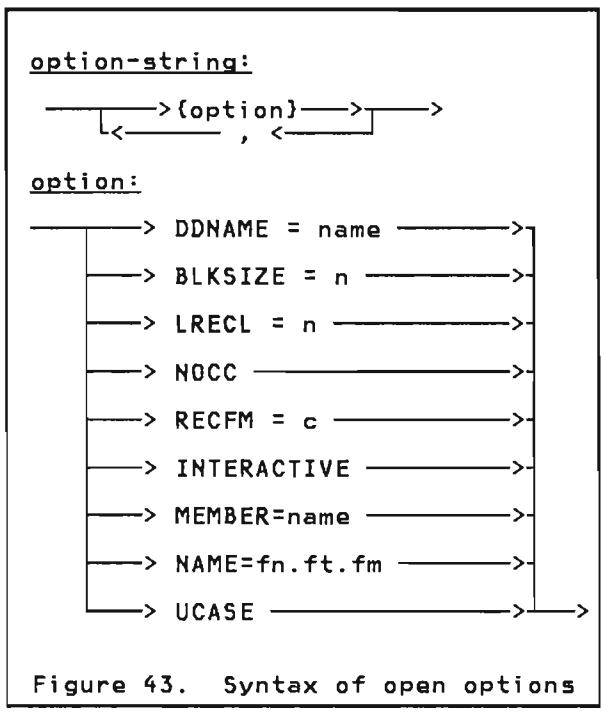


Figure 43. Syntax of open options

Not all of these options apply to all open procedures. If the option is specified for a procedure that is not applicable, the option will be ignored.

The following is a description of each option and the context in which it applies.

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 a file open routine with the **DDNAME** attribute specified.

If this option is not specified, then the DDname to be associated with the file is derived according to the following rules:

- If the file variable is a simple variable then the default DDname will be the name of the variable itself, truncated to 8 characters.
- If the file variable is an element of an array, a field of a record, or is pointer qualified, then a DDname will be generated of the following form: **PASCAL**nn, where "nn" is a two digit integer.

The **DDNAME** option is applicable to the following procedures:

RESET, REWRITE, UPDATE, PDSIN, and PDSOUT.

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.

This option is applicable to the procedure REWRITE only.

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.

For files with variable length records (RECFM=V), the logical record length must include a 4 byte length descriptor[10]. Thus, if text is being written to such a file, the LRECL must be 4 bytes longer than the longest line to be written.

The LRECL attribute may also be used in the TERMIN and TERMOUT procedures to specify the length of the I/O buffer. (This will determine the maximum length of the line to be read from, or written to, your terminal.)

This option is applicable to the procedures REWRITE, TERMIN, and TERMOUT.

NOCC
Normally, the first character position of an output file which contains ANSI or machine control characters (as determined by the RECFM) is not directly accessible to the user program. The data in such files is placed at the second character position of each record.

The NOCC option causes such files to be treated as though control characters are not significant; that is, data will be placed within each record at the first character position. This option allows control characters to be generated explicitly.

This option is applicable to the procedure REWRITE.

RECFM=c
This attribute is used to specify a record format to be associated with

[10] The 4 byte length descriptor for each record of a V-record file is an OS convention.

an output file. This specification (indicated by "**C**") will override a RECFM specification on the DDname definition.

Pascal/VS supports all record formats that QSAM supports:

```
U [T] [A]
      [M]

      [B ]
      [S ]
F     [T ]  [A]
      [BS]  [M]
V     [BT]
      [BST]

D [B] [A]
```

For an explanation of each of these record formats, consult the publication OS/VS2 MVS Data Management Services Guide (order number GC26-3875).

The **RECFM** specification applies to procedure REWRITE.

INTERACTIVE
This attribute indicates that the file is to be opened for input as an interactive file. See "Opening a File for Interactive Input" on page 46 for a description of interactive files.

This option applies to the procedures RESET and PDSIN. (This attribute is implied for TERMIN.)

MEMBER=name
This attribute specifies a member name of a partitioned data set (PDS). The member to be accessed is indicated by "name."

The **MEMBER** specification is required for the procedures PDSIN and PDSOUT (see "Partitioned Data Sets" on page 56).

NAME=fn.ft.fm (CMS only)
This attribute specifies the name of a CMS file which is to associated with the file variable. This option has no affect if the program is not running under CMS.

"fn," "ft," "fm" are the file name, file type and file mode, respectively, of the CMS file. Each must be separated by a period ("."). A file mode of "*" is permitted.

The **NAME** specification is applicable to the following procedures: RESET, REWRITE, UPDATE, PDSIN, and PDSOUT.

UCASE (CMS only)
This option causes text that is being read from a file opened by TERMIN to be translated to upper case. This option applies only to programs running under CMS; it is ignored otherwise.

```
program EXAMPLE;
var
  PDS    : TEXT;
  MEMBER : STRING(8);
  BUF    : packed array[1..80] of CHAR;
begin
  RESET(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      *)
      PDSIN(PDS,'DDNAME=SYSLIB,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 44. Using the open options

8.19 APPENDING TO A FILE

Data may be appended to an existing file by opening it for output with a call to REWRITE and specifying a disposition of "MOD" on the corresponding DDname definition.

The following examples illustrate how such a disposition is specified under the various operating system environments. The DDname of the file is "LOG"; the file name is "LOG.DATA."

```
CMS:
  FILEDEF LOG DISK LOG DATA (DISP MOD

TSO:
  ALLOC DDN(LOG) DSN(LOG.DATA) MOD

OS Batch:
  //LOG  DD DSN=ABC.LOG.DATA,DISP=MOD
```

9.1 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 45 on page 62. 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 46 on page 62. 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 31 for a description of compiler generated checks.) Figure 47 on page 62 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 48 on page 62 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 called routines
Routine           stmt at address in module
TRACE               4     02028C     AMPXSENV
HASHKEY             9     02018C     HASHCSEG
GET_HASH_PTR        2     021208     HASHBSEG
SEARCH_ID           9     0213C8     HASHBSEG
READ_ID             3     021550     HASHBSEG
<MAIN-PROGRAM>     24     020278     HASHASEG
PASCAL/VS                 02048C
```

Figure 45. Trace called by a user program

```
AMPX018E Fixed Point Overflow
     Trace back of called routines
Routine           stmt at address in module
FACTORIAL           3     02014C     TEST
FACTORIAL           3     02014C     TEST
FACTORIAL           3     02014C     TEST
<MAIN-PROGRAM>     17     020298     TEST
PASCAL/VS                 02048C
```

Figure 46. Trace call due to program error

```
AMPX032E High Bound Checking Error
     Trace back of called routines
Routine           stmt at address in module
TRANSLATE           4     020154     CONVERT
TO_ASCII           10     02024C     CONVERT
<MAIN-PROGRAM>     17     020338     CONVERT
PASCAL/VS                 02048C
```

Figure 47. Trace call due to checking error

```
 AMPX0401S File could not be opened: SYSIN
     Trace back of called routines
Routine           stmt at address in module
INITIALIZE          3     020154     COPY
<MAIN-PROGRAM>      2     020218     COPY
PASCAL/VS                 02048C
```

Figure 48. Trace call due to I/O error

9.2 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 the value of an array subscript, or the value being assigned to a subrange type variable is less than the minimum allowed for the subscript or subrange.

High bound
Either the value of an array subscript, or the value being assigned to a subrange type variable is greater than the maximum allowed for the subscript or subrange.

Nil pointer
an attempt was made to reference a variable from a pointer which has the value **nil**.

Case label
the expression of a **case**-statement has a value other than any of the specified **case** labels and there is no **otherwise** clause.

String truncation
the concatenation of two strings results in a string greater than 32767 characters in length, or there was an attempt to assign to a string a value which has more characters than the maximum length of the string.

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

String subscript out of bounds
there was an indexing operation on a string which was greater than the current length of the string.

Function value
a function routine returned to its invoker without being assigned a result.

9.3 EXECUTION ERROR HANDLING

Pascal/VS detects many kinds of errors during program execution; upon detection of an error, the Pascal/VS runtime library will provide error handling.

Certain errors are considered fatal by the runtime library. Examples of these errors are operation exception and protection exception. When a fatal error occurs the following happens:

1. Pascal/VS produces a message describing the error; the message is displayed on your terminal if you are executing in VM/CMS or TSO, or written to DDname SYSPRINT otherwise.
2. A trace back is displayed.
3. The program execution is terminated.

Other errors such as checking errors will not stop program execution. You must determine the extent to which the non-fatal errors affect your program results. Pascal/VS performs the following actions when a non-fatal error occurs.

1. A message describing the error is produced; the message is displayed on your terminal if you are executing in VM/CMS or TSO, or written to DDname SYSPRINT otherwise.
2. A trace back is generated.
3. If the program was compiled and linked with the 'DEBUG' option and the program was not executed with the 'DEBUG' run time option, then a symbolic dump of the variables in the procedure experiencing the error will be produced; the dump is displayed on your terminal if you are executing in VM/CMS or TSO, or written to DDname SYSPRINT otherwise.
4. If the program was compiled and linked with the 'DEBUG' option and the program was executed with the 'DEBUG' run time option then the interactive symbolic debugger will be invoked as if a breakpoint had been encountered.

Pascal/VS will allow a specific number of non-fatal errors to occur before the program is terminated. This number is set by the **ERRCOUNT** run time option (see "Run Time Options" on page 35). The default is 20.

```
(*****************************************************************)
(*                                                               *)
(* RUNTIME ERROR INTERCEPTION ROUTINE                            *)
(*                                                               *)
(*****************************************************************)

type
  ERRORTYPE = 1 .. 90;                  (*number of execution errors  *)
  ERRORACTIONS = (                      (*action to be performed      *)
      XHALT,                            (*terminate program           *)
      XPMSG,                            (*print pascal diagnostic    .*)
      XUMSG,                            (*print user's message        *)
      XTRACE,                           (*produce a trace back        *)
      XDEBUG,                           (*invoke the debugger         *)
      XDECERR,                          (*decr error counter          *)
      XRESERVED6,                       (*RESERVED                    *)
      XRESERVED7,                       (*RESERVED                    *)
      XRESERVED8,                       (*RESERVED                    *)
      XRESERVED9,                       (*RESERVED                    *)
      XRESERVEDA,                       (*RESERVED                    *)
      XRESERVEDB,                       (*RESERVED                    *)
      XRESERVEDC,                       (*RESERVED                    *)
      XRESERVEDD,                       (*RESERVED                    *)
      XRESERVEDE,                       (*RESERVED                    *)
      XRESERVEDF);                      (*RESERVED                    *)

  ERRORSET = set of ERRORACTIONS;

procedure ONERROR(
    const FERROR    : ERRORTYPE;        (*ERROR NUMBER                 *)
    const FMODNAME  : ALPHA;            (*MODULE NAME WHERE OCCURRED   *)
    const FPROCNAME : ALPHA;            (*PROCEDURE WHERE OCCURRED     *)
    const FSTMTNO   : INTEGER;          (*STATEMENT NO                 *)
    var   FRETMSG   : STRING;           (*RETURNED USER'S MESSAGE      *)
    var   FACTION   : ERRORSET);        (*ACTIONS TO BE PERFORMED      *)
EXTERNAL;
```

Figure 49. Contents of '%INCLUDE ONERROR'

Pascal/VS provides a mechanism for you to gain control when an execution time error occurs. When such an error occurs, a procedure called 'ONERROR' is called to perform any necessary action prior to generating a diagnostic. A default ONERROR routine is provided in the Pascal/VS library which does nothing.

You may write your own version of ONERROR and declare it as an EXTERNAL procedure. The procedure will be invoked when an error occurs; thus you may decide how the error should be handled. Figure 49 shows the contents of the IBM-supplied include file that contains the information relevant to producing your own ONERROR routine.

Upon entry to ONERROR the parameter FERROR contains the number of the error that has been encountered. See "Execution Time Messages" on page 154 to determine the message number corresponding to a particular error.[11]

FMODNAME, FPROCNAME, and FSTMTNO contain the name of the module, the name of the routine, and the source statement number, respectively, of the location where the error occurred.

FACTION is a set variable which determines what action is to be taken. Upon invocation of ONERROR, FACTION will describe the default action that will take place after ONERROR returns. You should examine this information and decide whether you would like to handle the error or let the default action take place.

[11] Each error intercepted by the Pascal/VS run time environment consists of a unique 3 digit number. A diagnostic message corresponding to the error will begin with the error number prefixed with the characters AMPX and suffixed with the character 'I', 'E' or 'S' (Informational, Error, Severe error).

You may modify the FACTION parameter as you desire. If you set the XUMSG member of FACTION then you must also set FRETMSG with the text of the message.

Figure 50 on page 65 is an example of a user interception of execution time errors.

```
% INCLUDE ONERROR;
procedure ONERROR;
begin
  (*do nothing if fixed, decimal or floating divide by zero *)
  (*and diagnose fixed-point overflow in procedure HASHFNC  *)
  if FERROR in [19, 21, 25] then
    FACTION := [ ]
  else
    if (FERROR = 18) & (FPROCNAME = 'HASHFNC') then
      begin
        FACTION := [XUMSG];
        FRETMSG := 'INPUT DATA CONTAINS GARBAGE';
      end;
end;
```

Figure 50. Example of User Error Handling

9.5 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 variable dump is placed on your terminal if you are executing in VM/CMS or TSO, or written to DDname SYSPRINT otherwise.

The Pascal/VS interactive debugger 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.

Under TSO and CMS, debugger commands are read directly from your terminal; likewise, the output is written directly to your terminal. If the debugger is being run in OS batch, then the input is read from DDname SYSIN; the output is sent to SYSPRINT.

In order to use the debugger, you must follow these three 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).[12]
- When executing the load module, specify 'DEBUG' as a run time option.[13] This will cause the debug environment to become active and you will be immediately prompted for a debugger command.

In the debugger environment the user may issue debug commands and examine variables in those modules which were compiled with the DEBUG option.

10.1 QUALIFICATION

A qualification consists of a module name and a routine name. The debugger 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.

10.2 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**, **END** and **CLEAR** commands which have no abbreviations. Square brackets ('[' and ']') are used in the command description to indicate optional parts of the command.

Semicolons are used to separate multiple commands on each line.

---

[12] 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 12.)

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 18.)

[13] Run time options must be terminated with a slash ('/'). See "Run Time Options" on page 35.

10.2.1 BREAK Command

Command Format:

BREAK [[module/] [routine]/] [stmt / END]

B [[module/] [routine]/] [stmt / END]

B

Where:

module is the name of a Pascal/VS module.
routine is the name of a procedure or function in the module.
stmt is a number of a statement in the designated routine.
END is a keyword which denotes the end of the 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. **stmt** 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. **END** specifies that the breakpoint is to occur in the epilogue of the routine immediately prior to the routine's return.

A maximum of 32 breakpoints may be set at any one time. The following table illustrates the meaning of the various forms.

Input	Module	Procedure
B S	current	current
B /S	current	main program
B P/S	current	P
B M//S	M	main program
B M/P/S	M	P

Where:
- current - means currently qualified module or procedure,
- M,P - are the names of a module or procedure
- S - is either a statement number or END

10.2.2 CLEAR Command

Command Format:

CLEAR

Minimum Abbreviation:

CLEAR

There are no operands.

The CLEAR command is used to remove all breakpoints.

10.2.3 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.

10.2.4 DISPLAY Command

Command Format:

DISPLAY

Minimum Abbreviation:

D

The DISPLAY command is used to display information about the current debugger 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.

10.2.5 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.

10.2.6 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.

10.2.7 END Command

Command Format:

END

Minimum Abbreviation:

END

The **END** command causes the program to immediately terminate. This command is synonymous with **QUIT**.

10.2.8 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.

The **EQUATE** command equates an identifier name to a data string. When the identifier name appears in a command, it will be expanded inline prior to executing the command.

As an example, the command

```
EQUATE X ,B[I]
```

will cause the variable "B[I]" to be viewed when "X" is entered as a command. The commands

```
EQUATE Y R@.F[6].J
,B[Y]
```

will cause the variable "B[R@.F[6].J]" to be viewed.

A semicolon may not terminate the **EQUATE** command; a semicolon will be treated as part of the data string. For example, the command

```
EQUATE Z GO;LISTVARS
```

will cause the "GO" and "LISTVARS" commands to be executed in succession when "Z" is entered as a command.

An equate command may be used to redefine the meaning of a debugger command:[14]

```
EQUATE GO WALK
```

makes the command "GO" function as the command "WALK."

An equate command may be cancelled by equating the previously defined identifier to an empty data string:

```
EQUATE Z
```

[14] There is one exception: the name EQUATE may not be equated to a data string.

removes the symbol "Z" from the debugger's equate table.

Equates may be equated to strings which contain other equates. All substitution will take place after expansion. The commands

```
EQUATE A P@.I
EQUATE B ,XYZ[A]
```

will cause the symbol "B" to be expanded to ",XYZ[P@.I]."

10.2.9 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.

10.2.10 Help Command

Command Format:

?

Minimum Abbreviation:

?

There are no operands.

The Help command lists all Debug commands.

10.2.11 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.

10.2.12 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.

10.2.13 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.

10.2.14 RESET Command

Command Format:

```
RESET [[module/] [routine]/] [stmt]
                             [END]
```

Minimum Abbreviation:

```
R [[module/] [routine]/] [stmt]
                         [END]
```

Where:

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

10.2.15 SET ATTR Command

Command Format:

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

Minimum Abbreviation:

```
S A [ ON  ]
    [ 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.

10.2.16 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.

10.2.17 SET TRACE Command

Command Format:

```
SET TRACE [ ON                ]
          [ OFF               ]
          [ TO ddname         ]
```

Minimum Abbreviation:

```
S T [ ON                ]
    [ OFF               ]
    [ TO ddname         ]
```

Where:

ddname is the name of a DDname where the trace output is to be sent.

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.

The output from the program trace normally will go to your terminal, by using the TO option you may direct the output to a specific file.

10.2.18 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.

10.2.19 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]
```

If the variable being viewed has not been assigned a value then the results depend on the variable's type:

- If the variable is of a simple type (integer, char, real, etc.), then the word "uninitialized" will be printed.
- If the variable is of a structured type (array, record), then the contents will be printed in hexadecimal; each byte of the the variable which is uninitialized will have the value 'FE' (hexadecimal).

10.2.20 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
```

10.2.21 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.

10.3 DEBUG TERMINAL SESSION

```
      |program Primgen;
      |type
      |  PrimeRange = 1..100;                  (*Specify limits for the      *)
      |                                        (*  number of prime numbers   *)
      |var
      |  Prime       : array[ PrimeRange ] of Integer;
      |                                        (*This array stores the result*)
      |  NotUsed     : PrimeRange;             (*Used test preceeding primes *)
      |  SaveIndex   : PrimeRange;             (*Used to remember last used  *)
      |                                        (*  spot in Prime             *)
      |  TestNumber  : Integer;                (*Test value for primeness    *)
      |
      |function IsPrime( Testval : INTEGER) : BOOLEAN;
      |  var
      |    Quotient,                           (*Testval div prime           *)
      |    Remainder  : Integer;               (*Test value for primeness    *)
      |    PrimeIndex : PrimeRange;            (*Used test preceeding primes *)
      |  begin                                 (*IsPrime                     *)
      |
    1 |    PrimeIndex := Lowest(PrimeRange);   (*Test each previous prime    *)
      |    repeat                              (*Starting with the first one *)
    2 |      PrimeIndex := Succ(PrimeIndex);   (*Get next prime             *)
      |
      |      (*Compute relative primeness of Testval and a known prime      *)
    3 |      Quotient := Testval div Prime[PrimeIndex];
    4 |      Remainder := Testval - Quotient * Prime[PrimeIndex]
    5 |    until (Remainder=0) | (Quotient <= Prime[PrimeIndex]);
      |
    6 |    if Remainder = 0 then               (*If the number was divided by*)
    7 |      IsPrime := FALSE                  (*any known Prime, then this  *)
      |    else                                (*is not prime                *)
    8 |      IsPrime := TRUE;
      |  end;                                  (*IsPrime                     *)
      |
      |begin
    1 |  Prime[1]    := 2;                     (*First three primes          *)
    2 |  Prime[2]    := 3;                     (*     ditto                  *)
    3 |  Prime[3]    := 5;                     (*     ditto                  *)
    4 |  TestNumber  := 5;                     (*Start candidates at 5       *)
    5 |  SaveIndex   := 3;                     (*Last used prime entry       *)
      |
      |  repeat
    6 |    TestNumber := TestNumber + 2;       (*Test each odd number        *)
      |                                        (*  starting with the first   *)
    7 |    if IsPrime(TestNumber) then         (*If canidate is a prime      *)
      |      begin                            (*Save it in the next entry   *)
    8 |        SaveIndex:= Succ(SaveIndex);    (*  of the prime table       *)
    9 |        Prime[SaveIndex] := TestNumber
      |      end
   10 |  until SaveIndex = Highest(PrimeRange);
      |
      |  (*Print results at ten to a line                                    *)
   11 |  for PrimeIndex := Lowest(PrimeRange) to Highest(PrimeRange) do
      |    begin
   12 |      Write( Prime[PrimeIndex]:7  );    (*Print one prime number      *)
   13 |      if (PrimeIndex mod 10) = 0 then   (*If ten have been printed    *)
      |        Writeln                         (*  then skip to next line   *)
   14 |    end;
      |
      |end.                                    (*Primgen                     *)
```

Figure 51. Sample program for Debug session

The following series of figures is a sample Debug terminal session that demonstrates breakpoints, viewing variables and other DEBUG commands. User commands are **highlighted and underlined**. The program being executed is shown in Figure 51.

```
pascalvs primgen (debug
 INVOKING PASCAL/VS R2.0
 NO COMPILER DETECTED ERRORS

 Source lines:     62;  Total time:    1.20 seconds;  Total rate:    3092 LPM
R; T=1.73/3.05 16:13:54


pascmod primgen (debug
R; T=0.90/2.19 16:14:51


filedef output terminal
R; T=0.03/0.05 16:14:52


primgen debug count /
 Debug(PRIMGEN <MAIN-PROGRAM>):
```

Figure 52. Compiling, linking and executing a program with DEBUG

```
?
 Name (abbreviation is in capital letters)
  ?               This command list
  ,               Display a variable
  Break           Set a breakpoint
  CLEAR           Remove all breakpoints
  Cms             Enter CMS subset mode
  Display         Display current resume point
  Display Break   Display currently set breakpoints
  Display Equate  Display currently set equates
  END             Halt your program
  Equate          Set an identifier to a literal value
  Go              Continue executing your program
  Listvars        List all variables
  Qual            Set default module/routine
  QUIT            Halt your program
  Reset           Remove a specific breakpoint
  Set Attr        Set default viewing information ON/OFF
  Set Count       Turn statement counting ON/OFF
  Set Trace       Turn tracing ON/OFF/TO fileid
  Trace           Display invocation chain of routines
  Walk            Execute one statement of current routine
 Debug(PRIMGEN <MAIN-PROGRAM>):
```

Figure 53. The HELP command of DEBUG

```
break 8
 PRIMGEN/<MAIN-PROGRAM>/8
 Debug(PRIMGEN <MAIN-PROGRAM>):


go
 Stopped at PRIMGEN/<MAIN-PROGRAM>/8
 Debug(PRIMGEN <MAIN-PROGRAM>):


walk
 Stopped at PRIMGEN/<MAIN-PROGRAM>/9
 Debug(PRIMGEN <MAIN-PROGRAM>):


walk
 Stopped at PRIMGEN/<MAIN-PROGRAM>/10
 Debug(PRIMGEN <MAIN-PROGRAM>):
```

Figure 54. Setting Breakpoints and Statement Walking

```
listvars
 Variables for procedure: <MAIN-PROGRAM>
  PRIME
 (0003CA28)
 000000 00000002 00000003 00000005 FEFEFEFE '................'
 000010 FEFEFEFE FEFEFEFE FEFEFEFE FEFEFEFE '................'
 (00000020 through 0000018F is the same as above)
  NOTUSED =  uninitialized
  SAVEINDEX = 3
  TESTNUMBER = 7
 Debug(PRIMGEN <MAIN-PROGRAM>):
```

Figure 55. The LISTVARS command - List all variables

```
set trace on
 Program trace is on -- output to '<TERMINAL>'
 Debug(PRIMGEN <MAIN-PROGRAM>):


go
 Resuming  PRIMGEN  <MAIN-PROGRAM>
 ======>   6-7
 Executing PRIMGEN  ISPRIME
 ======>   1
 ======>   2-5
 ======>   6
 ======>   7
 Returning from ISPRIME
 Resuming  PRIMGEN  <MAIN-PROGRAM>
 ======>  10
 ======>   6-7
 Executing PRIMGEN  ISPRIME
 ======>   1
 ======>   2-5
 ======>   6
 ======>   8
 Returning from ISPRIME
 Resuming  PRIMGEN  <MAIN-PROGRAM>
 ======>   8-9
 Stopped at PRIMGEN/<MAIN-PROGRAM>/8
 Debug(PRIMGEN <MAIN-PROGRAM>):
```

Figure 56. The Trace Mode of DEBUG

```
go
 ======>  10
 ======>   6-7
 Executing PRIMGEN  ISPRIME
 ======>   1
 ======>   2-5
 ======>   2-5
 ======>   6
 ======>   8
 Returning from ISPRIME
 Resuming  PRIMGEN  <MAIN-PROGRAM>
 ======>   8-9
 Stopped at PRIMGEN/<MAIN-PROGRAM>/8
 Debug(PRIMGEN <MAIN-PROGRAM>):


walk
 Stopped at PRIMGEN/<MAIN-PROGRAM>/9
 Debug(PRIMGEN <MAIN-PROGRAM>):


walk
 ======>  10
 Stopped at PRIMGEN/<MAIN-PROGRAM>/10
 Debug(PRIMGEN <MAIN-PROGRAM>):


walk
 ======>   6-7
 Stopped at PRIMGEN/<MAIN-PROGRAM>/6
 Debug(PRIMGEN <MAIN-PROGRAM>):


walk
 Stopped at PRIMGEN/<MAIN-PROGRAM>/7
 Debug(PRIMGEN <MAIN-PROGRAM>):


walk
 Executing PRIMGEN  ISPRIME
 ======>   1
 ======>   2-5
 ======>   6
 ======>   7
 Returning from ISPRIME
 Resuming  PRIMGEN  <MAIN-PROGRAM>
 ======>  10
 Stopped at PRIMGEN/<MAIN-PROGRAM>/10
 Debug(PRIMGEN <MAIN-PROGRAM>):


go
 Stopped at PRIMGEN/<MAIN-PROGRAM>/8
 Debug(PRIMGEN <MAIN-PROGRAM>):
```

Figure 57. Walking when the Trace Mode is On

```
display qualification
 Currently qualified to PRIMGEN <MAIN-PROGRAM>
 Will resume at PRIMGEN <MAIN-PROGRAM> 8
 Counts are on
 Trace is on
 Trace output to <TERMINAL>
 Debug(PRIMGEN <MAIN-PROGRAM>):


display breaks
 Module            Routine           Stmt
 PRIMGEN           <MAIN-PROGRAM>     8
 Debug(PRIMGEN <MAIN-PROGRAM>):


equate tn ,testnumber
 Debug(PRIMGEN <MAIN-PROGRAM>):


tn
,TESTNUMBER
  TESTNUMBER = 19
 Debug(PRIMGEN <MAIN-PROGRAM>):



display equate
 TN                ==> ,TESTNUMBER
 Debug(PRIMGEN <MAIN-PROGRAM>):


set trace off
 Program trace is off
 Debug(PRIMGEN <MAIN-PROGRAM>):
```

Figure 58. Miscellaneous DEBUG Commands

```
,testnumber
  TESTNUMBER = 19
 Debug(PRIMGEN <MAIN-PROGRAM>):


, testnumber (attr
     DATA TYPE: INTEGER
 MEMORY CLASS : LOCAL AUTOMATIC
 DECLARED IN  : <MAIN-PROGRAM>
  TESTNUMBER  = 19
 Debug(PRIMGEN <MAIN-PROGRAM>):


,prime[10]
  PRIME[10] =  uninitialized
 Debug(PRIMGEN <MAIN-PROGRAM>):


,prime[5]
  PRIME[5] = 11
 Debug(PRIMGEN <MAIN-PROGRAM>):
```

Figure 59. Commands to Display a Variable

```
break isprime/end
 PRIMGEN/ISPRIME/END
 Debug(PRIMGEN <MAIN-PROGRAM>):


go
 Stopped at PRIMGEN/ISPRIME/END
 Debug(PRIMGEN ISPRIME):


trace
      Trace back of called routines
 Routine            stmt at address in module
 ISPRIME              8      020138      PRIMGEN
 <MAIN-PROGRAM>       7      020260      PRIMGEN
 PASCAL/VS                   02055A

 Debug(PRIMGEN ISPRIME):


set trace on
Program trace in on -- output to '<TERMINAL>'
 Debug(PRIMGEN ISPRIME):


equate next go;listvars
 Debug(PRIMGEN ISPRIME):


next
GO;LISTVARS
 Resuming  PRIMGEN  <MAIN-PROGRAM>
 ======>   8-9
 ======>  10
 ======>   6-7
 Executing PRIMGEN  ISPRIME
 ======>   1
 ======>   2-5
 ======>   6
 ======>   7
 Returning from ISPRIME
 Stopped at PRIMGEN/ISPRIME/END
 Variables for procedure: ISPRIME
  PRIMEINDEX = 2
  QUOTIENT = 13
  REMAINDER = 0
  TESTVAL = 39
 Debug(PRIMGEN ISPRIME):


set trace off
Program trace is off
 Debug(PRIMGEN <MAIN-PROGRAM>):
```

Figure 60. Using Multiple commands on one Line and other commands

```
reset 8
 Breakpoint at PRIMGEN/<MAIN-PROGRAM>/8 has been removed
 Debug(PRIMGEN <MAIN-PROGRAM>):


go
 Stopped at PRIMGEN/ISPRIME/END
 Debug(PRIMGEN ISPRIME):


listvars
 Variables for procedure: ISPRIME
  PRIMEINDEX = 2
  QUOTIENT = 11
  REMAINDER = 0
  TESTVAL = 33
 Debug(PRIMGEN ISPRIME):


reset end
 Breakpoint at PRIMGEN/ISPRIME/END has been removed
 Debug(PRIMGEN ISPRIME):


go
      2       3       5       7      11      13      17      19      23      29
     31      37      41      43      47      53      59      61      67      71
     73      79      83      89      97     101     103     107     109     113
    127     131     137     139     149     151     157     163     167     173
    179     181     191     193     197     199     211     223     227     229
    233     239     241     251     257     263     269     271     277     281
    283     293     307     311     313     317     331     337     347     349
    353     359     367     373     379     383     389     397     401     409
    419     421     431     433     439     443     449     457     461     463
    467     479     487     491     499     503     509     521     523     541
```

Figure 61. The Reset Breakpoint Command

```
 PASCAL/VS  STATEMENT COUNTING SUMMARY                        PAGE 1

 <MAIN-PROGRAM> IN PRIMGEN CALLED 1 TIME(S)
 FROM-TO:COUNT FROM-TO:COUNT FROM-TO:COUNT FROM-TO:COUNT
   1-5  :1          6-7  :268         8-9  :97           10 :268
     11 :1         12-13 :100          14 :10

 ISPRIME IN PRIMGEN CALLED 268 TIME(S)
 FROM-TO:COUNT FROM-TO:COUNT FROM-TO:COUNT FROM-TO:COUNT
     1  :268         2-5  :910          6  :268          7  :171
     8  :97
```

Figure 62. Statement Counting Summary

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

11.1 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 invocation of a routine 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.

11.2 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.

11.3 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.[15] The name of the section is derived from the first eight characters of the variable's name.

11.4 DYNAMIC STORAGE

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

11.5 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.

11.6 DATA SIZE AND BOUNDARY ALIGNMENT

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

11.6.1 The Predefined Types

The table in Figure 63 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
SHORTREAL	4	FULL WORD
REAL	8	DOUBLE WORD
STRING(len)	len+2	HALF WORD
STRINGPTR	8	FULL WORD

Figure 63. Storage mapping for predefined types

[15] Each **def** variable becomes a named COMMON block which may be used to communicate with FORTRAN subroutines.

11.6.2 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.

11.6.3 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 64. 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
0..16777215	3	BYTE
-8388608..8388607	3	BYTE
otherwise	4	FULL WORD

Figure 64. 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.

11.6.4 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 65. Alignment of records

Packed records are always aligned on a byte boundary.

11.6.5 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
```

11.6.6 FILEs

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

11.6.7 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 66 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	BYTE
25 <= M <= 32	4	FULL WORD
33 <= M <= 256	(M+7) div 8	BYTE

Figure 66. Storage mapping of SETs

11.6.8 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;
```

12.1 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.

12.2 REGISTER USAGE

The table in Figure 67 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 the DSA of the main program
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

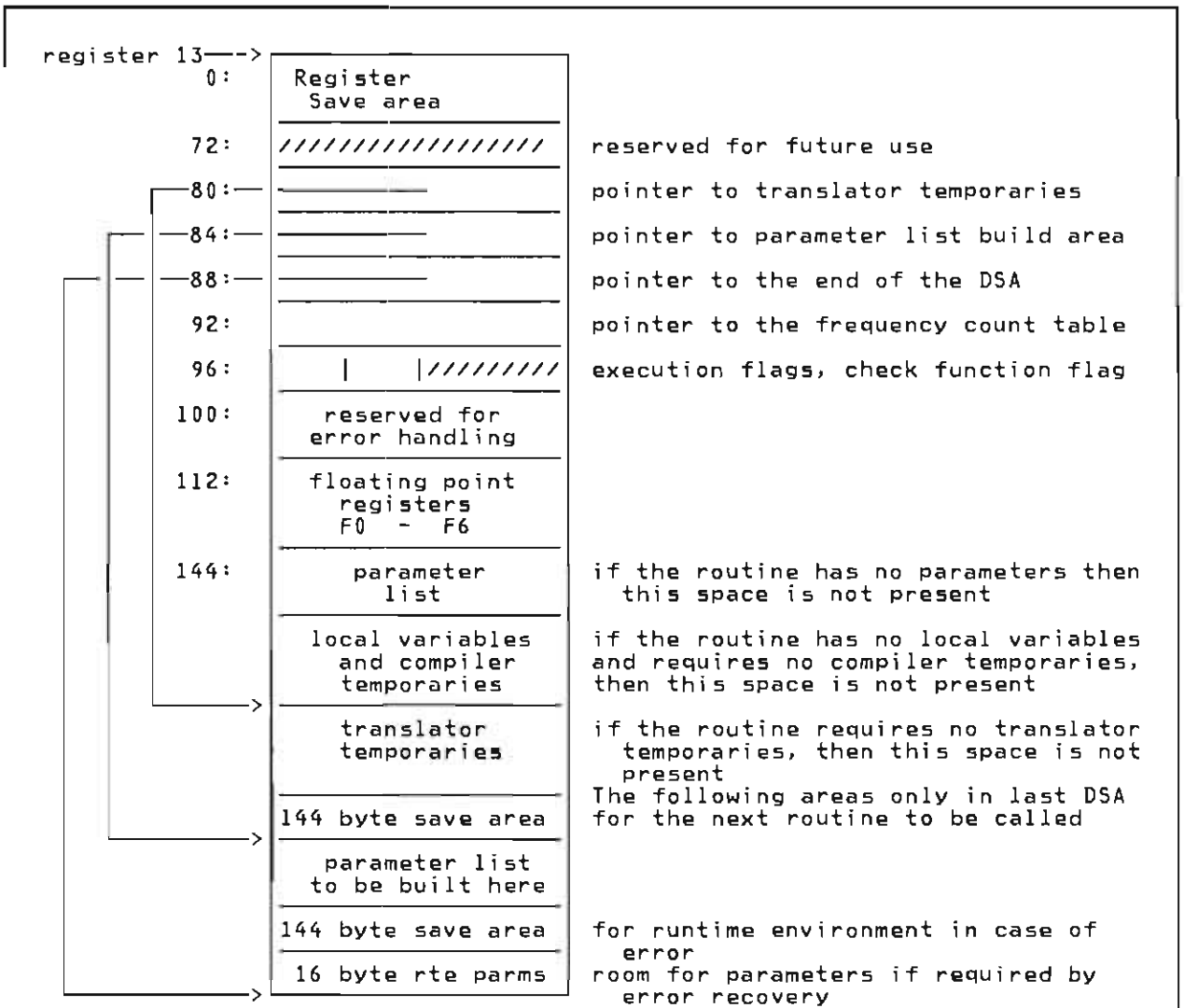
Figure 67. Register usage

12.3 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. A Pascal/VS routine requires a DSA of at least 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.

Figure 68 illustrates the structure of the DSA. Figure 69 on page 95 shows the DSECT expansion of the DSA. (A copy of this DSECT may be found in member DSA of the standard include library[16].)



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

Figure 68. DSA format

[16] Under MVS, the name of this library is sys1.**PASCALVS.MACLIB**. Under CMS, it is PASCALVS MACLIB.

```
DSA       DSECT
DSASDIS   DS     F          Save space for display level
DSALSVA   DS     F          Pointer to last save area
          DS     F          (reserved for future use)
DSARETA   DS     F          Return address
DSAEPAD   DS     F          Entry point address
DSARG0    DS     F          Save area for register 0
DSAPREG   DS     F          Save area for parameter list pointer (reg 1)
DSACODE   DS     F          Save area for base register for code (reg 2)
DSARG3    DS     F          Save area for register 3
DSARG4    DS     F          Save area for register 4
DSARG5    DS     F          Save area for register 5
DSARG6    DS     F          Save area for register 6
DSARG7    DS     F          Save area for register 7
DSARG8    DS     F          Save area for register 8
DSARG9    DS     F          Save area for register 9
DSACOD2   DS     F          Save area for 2nd base register for code (reg 10)
DSAL1B    DS     F          Save area for register 11 (main DSA address)
DSAPCWA   DS     F          Save area for register 12 (PCWA pointer)
DSAAKEY   DS     F          Used by attention processor
DSARES4   DS     F          Reserved
DSATPTR   DS     F          Address of temporary section of DSA
DSAPPTR   DS     F          Address of parameter list build area
DSARPTR   DS     F          Address of runtime parameter list build area
DSACNTS   DS     F          Address of count table
DSARAID   DS     X          Interactive debugger flags
DSAFUNX   DS     X          Function assignment check flag
DSARES1   DS     2X         Reserved
DSACKSA1  DS     F          Save area utilized by error recovery
DSACKSA2  DS     F          Save area utilized by error recovery
DSACKSA3  DS     F          Save area utilized by error recovery
DSAFL0    DS     D          Save area for floating point register 0
DSAFL2    DS     D          Save area for floating point register 2
DSAFL4    DS     D          Save area for floating point register 4
DSAFL6    DS     D          Save area for floating point register 6
DSALEN    EQU    *-DSA      Length of DSA header
          SPACE  1
DSAPRM1   DS     F          Start of parameters and/or local variables
DSAPRM2   DS     F
DSAPRM3   DS     F
DSAPRM4   DS     F
DSAPRM5   DS     0F
DSADATA   DS     F
```

Figure 69. DSA DSECT: anchored off of register 13.

12.4 ROUTINE INVOCATION

Each invocation of a Pascal/VS routine must acquire a dynamic storage area (DSA) (see "Dynamic Storage Area" on page 94). 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 70 illustrates the condition of the stack and relevant registers immediately at the start of a routine.

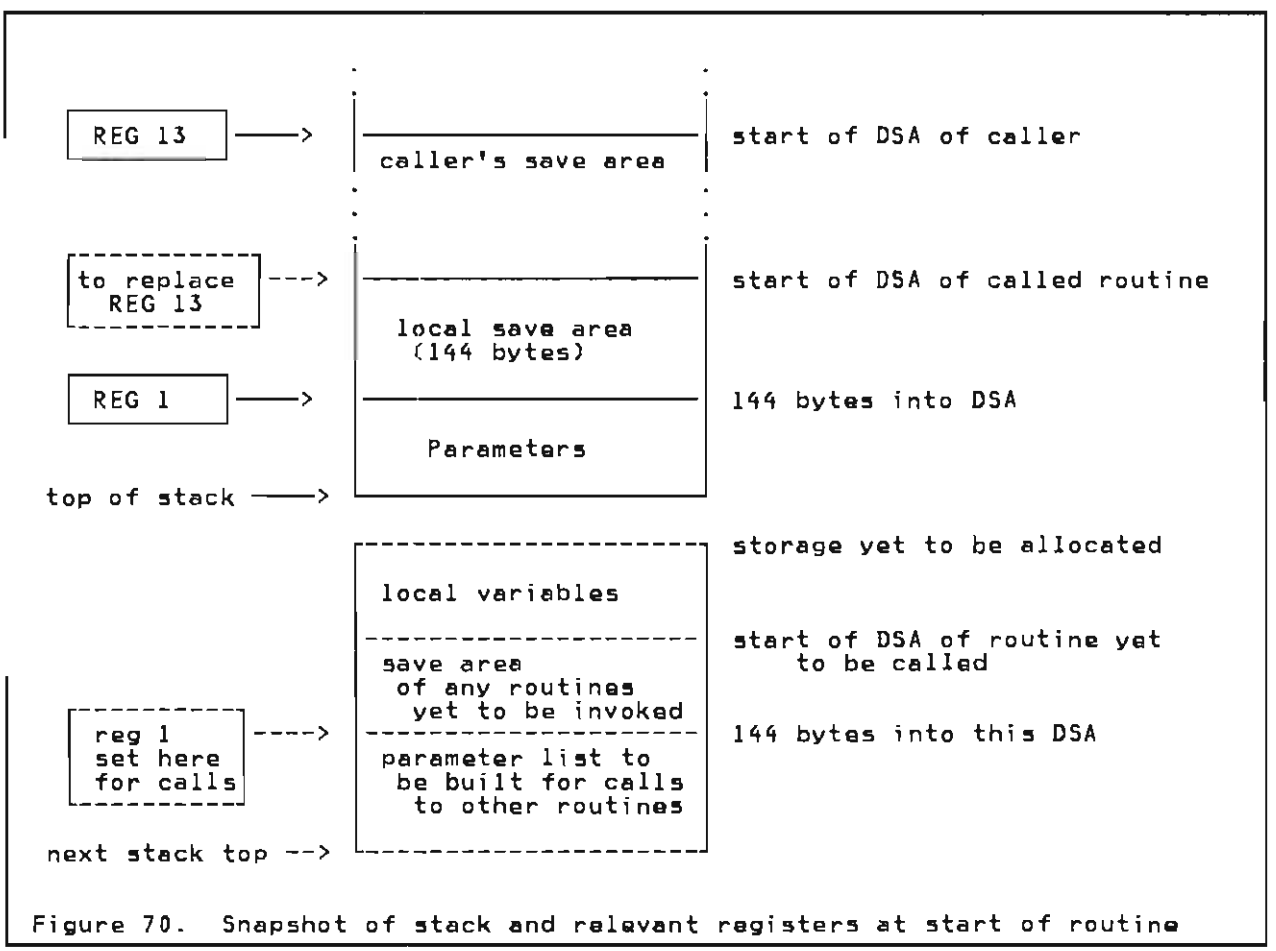


Figure 70. Snapshot of stack and relevant registers at start of routine

12.5 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.

12.5.1 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 71. Passing by Read/Write reference

12.5.2 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 72. Passing by Read-only reference

12.5.3 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 73. Passing by value

12.5.4 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 74. Passing routine parameters

12.5.5 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 75. Function results

12.6 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.

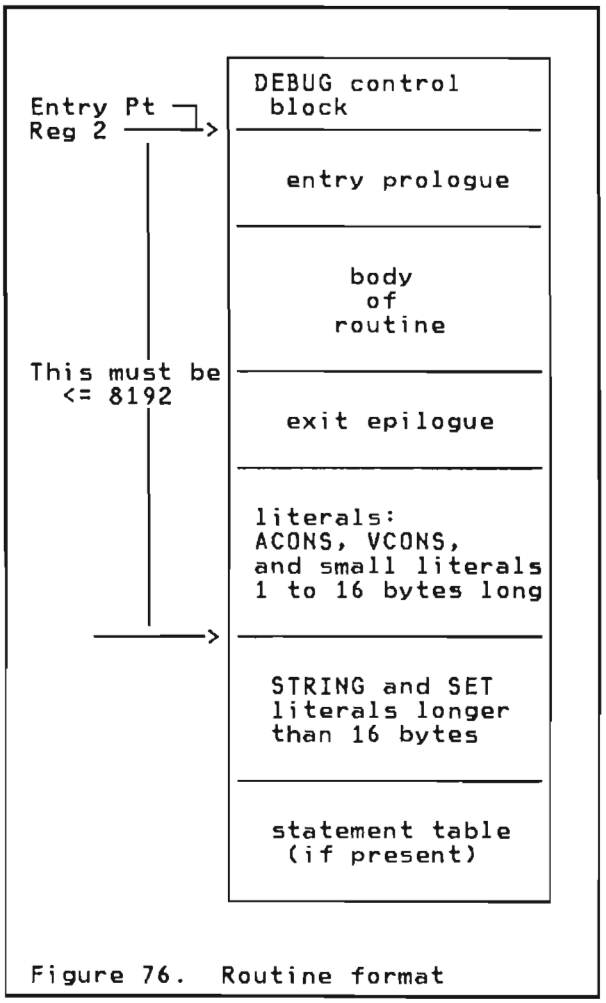


Figure 76. Routine format

```
PCWA     =
  record
    PCWAENDS        : INTEGER;          (*Ptr to end of current stack   *)
    PCWACURS        : INTEGER;          (*Ptr to start of current stack *)
    PCWASELF        : INTEGER;          (*Self identifying flag 'PCWA'  *)
    PCWAFL2         : PCWA_FLG_SET;     (*compiler runtime flags        *)
    PCWARC(16)      : INTEGER;          (*Return code                   *)
    PCWAFILE        : PCBP;             (*pointer to open files         *)
    PCWAPARM        : SYSPARMP;         (*parms string                  *)
    PCWAMODS        : DBCBP;            (*module header chain (debugger)*)
    PCWAESAP        : INTEGER;          (*ptr to external save area     *)
    PCWADISP        : array[0..7] of DSAP;(*DISPLAY                     *)
    PCWADTMP        : INTEGER;          (*Debugger temporary            *)
    PCWARTMP        : REAL;             (*floating point temporary      *)
    PCWAR0          : REAL;             (*'4E00000000000000'X           *)
    PCWA2231        : REAL;             (*'4E00000010000000'X           *)
    PCWAMASK        : ALFA;             (*'8040201008040201'X           *)
    PCWAMFIX        : ALFA;             (*temp for first 8 bytes of DSA *)
    PCWASAVE        : array[1..36] of INTEGER; (*Extra save area        *)
    PCWAPLST        : array[1..16] of INTEGER; (*parm list build        *)
    PCWAFIN         : INTEGER;          (*Pointer to the HALT address  *)
    PCWAALLC        : INTEGER;          (*address of memory allocator  *)
    PCWADLLC        : INTEGER;          (*address of memory deallocator*)
    PCWASDFT        : INTEGER;          (*default stack size            *)
    PCWACHKR        : INTEGER;          (*address of checker routine    *)
    PCWADSAS        : INTEGER;          (*size of DSA in bytes (144)    *)
    PCWAMEMF        : INTEGER;          (*addr of memory fixup routine  *)
    PCWAFLAG        : INTEGER;          (*Inter-language communication  *)
    PCWAPICA        : ALFA;             (*PICA save area                *)
    PCWASEED        : INTEGER;          (*seed of 'RANDOM' function     *)
    PCWAXEND        : INTEGER;          (*end of stack for SETMEM       *)
    PCWAECNT        : INTEGER;          (*error count until abend       *)
    PCWACHK         : INTEGER;          (*address of check routine      *)
    PCWACMEM        : INTEGER;          (*current memory in use         *)
    PCWASTAX        : space[20] of CHAR;(*STAX list form                *)
    PCWAEOPN        : BOOLEAN;          (*TRUE if PCWAEOUT is open      *)
    PCWADINT        : BOOLEAN;          (*TRUE if debugger initialized  *)
    PCWATSO         : BOOLEAN;          (*TRUE if TSO environment       *)
                    : BOOLEAN;          (*reserved                      *)
    PCWAATTN        : INTEGER;          (*address of attn handling      *)
    PCWAFCNT        : INTEGER;          (*cnt of files without DDnames  *)
    PCWASIZE        : INTEGER;          (*size of initial alloc for pcwa*)
    PCWADINA        : INTEGER;          (*Address of AMPDINIT or nil    *)
    PCWABOPA        : INTEGER;          (*Address of AMPDIBOP or nil    *)
    PCWABBA         : INTEGER;          (*Address of AMPDIBB or nil     *)
    PCWAERAD        : INTEGER;          (*Error address - CHKR or DIAG  *)
    PCWAFSTK        : INTEGER;          (*Chain of free dsa stack elems *)
    PCWAENDA        : INTEGER;          (*Address of AMPDEPIL or nil    *)
    PCWAHDFT        : INTEGER;          (*default heap size             *)
    PCWAPROC(1200)  : space[64] of CHAR;(*Work area for PROCESS         *)
    PCWAUSER(1264)  : space[64] of CHAR;(*Area reserved for user        *)
    PCWAEOUT(1328)  : TEXT;             (*ERROR OUTPUT PCB              *)
    PCWAOUT(1392)   : PCB;              (*OUTPUT PCB                    *)
    PCWAIN(1456)    : PCB;              (*INPUT PCB                     *)
    PCWAPDAT(1520)  : STRING(254);      (*actual parm list after format *)
    PCWAERSA(1776)  : SPIEDSA;          (*savearea for error routines   *)
    PCWAPIE         : PSW;              (*PSW from PIE                  *)
    PCWASPIE        : INTEGER;
    PCWAMEMA(1984)
                    : array[MEM_LEVELS] of SPACE_DESC;
                                        (*space for memory allocator   *)
  end;
```

Figure 77. Pascal Communications Work Area

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 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 (such as string concatenation). Figure 77 on page 100 shows the structure of the first half of the PCWA. Each field is described below:

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

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

PCWASELF
a self defining field that is set to 'PCWA'.

PCWAFL2
flags used to enable runtime features.

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

PCWAFILE
a pointer to the first file (PCB) that has been opened but not closed.

PCWAPARM
a pointer to the parameter string passed to the program.

PCWAMODS
a pointer to the head of a chain that links modules together as required by the interactive debugger.

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

PCWADISP
the runtime display - a stack of 8 base registers that contains the address of the DSAs that are available to the executing routine.

PCWADTMP
a temporary used by the interactive debugger.

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

PCWAR0
a constant that contains the floating point value zero.

PCWA2231
a constant that contains the floating point value of 2 raised to the 31 power minus 1 in an unnormalized form.

PCWAMASK
eight bytes that contain masks which are used in set operations.

PCWAMFIX
a temporary used during runtime error recovery.

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

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

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

PCWAALLC
address of a system dependent routine which is responsible for allocating blocks of storage.

PCWADLLC
address of a system dependent routine which releases blocks of storage.

PCWASDFT
the number of bytes that the stack will be extended if an overflow should occur. (Set by the STACK run time option.)

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

PCWADSAS
the size of the smallest DSA. Its value is 144.

PCWAMEMF
contains the address of the memory fixup routine, which is called when the DSA stack overflows.

PCWAFLAG
a flag used when communicating between different languages.

PCWAPICA
is used for a save area for the PICA.

PCWASEED
contains the current seed for the RANDOM function.

PCWAXEND
contains the true end of the current stack. PCWAENDS may not be correct, PCWAENDS is made incorrect in order to force a call to

AMPXMEMF so that a DSA may be initialized (if SETMEM option is enabled).

PCWAECNT
contains the number of non-fatal errors which will be tolerated before the program will be abended.

PCWACHK
contains the address of the routine which gains control when a checking error occurs. This routine is normally AMPXCHKR.

PCWACMEM
defines which heap is in use, normally the value is one, which indicates that the users heap is available.

PCWASTAX
contains the list form of the STAX macro.

PCWAEOPN
a flag that indicates whether the error file, PCWAEOUT has been opened.

PCWADINT
is a flag indicating whether AMPDCOM (debugger common area) has been initialized yet.

PCWATSO
is a flag indicating whether we are executing in a TSO environment.

PCWAATTN
contains the address of the terminal attention routine.

PCWAFCNT
contains the number of the next generated DDname.

PCWASIZE
contains the size of the initial allocation of the PCWA.

PCWADINA
contains the address of the AMPDINIT routine, which initializes the interactive debugger.

PCWABOPA
contains the address of the AMPDIBOP routine, which is invoked at each procedure entry when the debugger is active.

PCWABBA
contains the address of the AMPDIBB routine, which is invoked at each basic block of code when the debugger is active.

PCWAERAD
contains the offending address when a checking error or a program error occurs.

PCWAFSTK
points to the beginning of a chain of all free blocks of storage.

PCWAENDA
address of the AMPDEPIL routine, which is invoked from the epilogue of each routine when the debugger is active.

PCWAHDFT
the number of bytes that the heap will be extended each time it overflows. (Set by the HEAP run time option.)

PCWAPROC
reserved for future use.

PCWAUSER
reserved for Pascal/VS users.

PCWAEOUT
the file (PCB) to where execute time error diagnostics is sent.

PCWAOUT
the PCB for the standard file OUTPUT.

PCWAIN
the PCB for the standard file INPUT.

PCWAPDAT
a string that contains the passed in symbolic parameter list after it it has been formatted.

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

PCWAPIE
a place to save certain information from the SPIE.

PCWASPIE
spie work area

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

12.8 PCB - PASCAL FILE CONTROL BLOCK

```
PCB   =                                  (*Pascal Control Block        *)
  record
    PCBFILEP :   BUFFERP;                (*file pointer                *)
    PCBFLAGS :   FILEFLAGS;              (*file flags                  *)
    PCBELEM  :   HALFWORD;               (*length of file component    *)
    PCBNAME  :   ALFA;                   (*file-variable name          *)
    PCBCODE  :   MagicNumber;            (*initialization test         *)
    PCBBUFIDX:   HALFWORD;               (*buffer index                *)
    PCBBUFLEN:   HALFWORD;               (*buffer length               *)
    PCBBUFP  :   BUFFERP;                (*pointer to start of buffer  *)
    PCBOPTP  :   OPTP;                   (*ptr to OPTIONs descriptor   *)
    PCBLAST  :   PCBP;                   (*link to last PCB of chain   *)
    PCBNEXT  :   PCBP;                   (*link to next PCB of chain   *)
    PCBICBP  :   ICBP;                   (*ptr to Implem. Ctrl Block   *)
    PCBSTART :   HALFWORD;               (*initial value of PCBBUFIDX  *)
    PCBSTAT  :   IOSTATUS;               (*status of last open         *)
             :   CHAR;                   (*<not-used>                  *)
             :   INTEGER;                (*<not-used>                  *)
             :   INTEGER;                (*<not-used>                  *)
             :   INTEGER;                (*<not-used>                  *)
             :   INTEGER;                (*<not-used>                  *)
  end;
```

Figure 78. Pascal file Control Block (PCB) format

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

The fields are defined as:

PCBFILEP
points to the current element of the file.

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

FINPUT indicates that file is open for input.

FOUTPUT indicates that file is open for output.

FTEXT the file is of type TEXT.

FEOLN end-of-line condition is true.

FEOF end-of-file condition is true.

FFIXED file has fixed length records.

FINTER the file was opened as an interactive file.

FSTATUS the user will check PCBSTAT and report the errors.

FFEOL end-of-line condition is true, but not as a result of READLN.

FOPTS an options string was specified in the last open.

FWRAP indicates that one or more lines of the text file (opened for output) has exceeded the logical record length of the file.

FERR indicates that a read was attempted after the end-of-file condition became true. This flag is used to suppress multiple error diagnostics from a single READ statement.

PCBELEM
the length of one component of the file.

PCBNAME
the DDNAME of the file.

PCBCODE
an encoded value that is used to test whether the PCB has been initialized; this is not required for files which are local variables but is needed for files that are allocated dynamically (NEW).

PCBBUFIDX
byte index into the I/O buffer (PCBBUFP).

PCBBUFLEN
total length of buffer in bytes.

PCBBUFP
address of the beginning of the buffer.

PCBOPTP
address of the control block that describes the information passed through the options string as the file is being opened. The procedures which open a file and pass an options string are: RESET, REWRITE, UPDATE, TERMIN, TERMOUT, PDSIN or PDSOUT.

PCBLAST
back chain of currently open PCBs.

PCBNEXT
forward chain of currently open PCBs.

PCBICBP
points to a system dependent control block to be used by the lowest level of interface to the I/O access methods.

PCBSTART
contains the initial value of PCBBUFIDX, which is used to determine if the current buffer contains any data that needs processing prior to closing the file.

PCBSTAT
status of the file.

It is sometimes desirable to invoke subprograms (procedures) written in other programming languages: this is useful to obtain services not available directly in Pascal/VS. It is also desirable to have a Pascal/VS procedure called from a non-Pascal program: this would allow you to take advantage of Pascal in an existing application without rewriting the entire application. This chapter will discuss the options available to you and what you must do in order to have this flexibility.

We can divide inter-language communication into two classes:

- The Pascal procedure is the calling procedure and the non-Pascal procedure is being called.
- The Pascal procedure is called from a non-Pascal calling procedure.

Your options are summarized in Figure 79.

	Pascal as the calling language	Pascal as the called language
FORTRAN	Define procedures and functions in Pascal using the FORTRAN directive. This enables you to call a subprogram written in FORTRAN.	Use a call statement in FORTRAN to call the Pascal procedure. The Pascal procedure must be defined with the MAIN directive. After the last call to a Pascal procedure you must call PSCLHX (Pascal halt execution).
Assembler	Define procedures and functions in Pascal using the FORTRAN or the EXTERNAL directive. If you use EXTERNAL you will be able to specify an arbitrary Pascal parameter list.	Use a V-type constant in the Assembler routine to define the Pascal entry point. You must define the Pascal procedure as EXTERNAL, MAIN, or REENTRANT. After the last call to a Pascal procedure you must call PSCLHX.
COBOL	Define procedures and functions in Pascal using the FORTRAN directive. This enables you to call a subprogram written in COBOL. You may desire to call ILBOSTP0 prior to calling a COBOL program. Consult the COBOL Programmer's guide for details.	Use a call statement in COBOL to call the Pascal procedure. COBOL should be compiled with the 'NODYNAM' option and the call must be a call of a literal. The Pascal procedure must be defined with the MAIN directive. After the last call to a Pascal procedure you must call PSCLHX.
PL/I	Define procedures and functions in Pascal using the FORTRAN directive. This enables you to call a subprogram written in PL/I. You should define the PL/I procedure with the FORTRAN option. Consult the PL/I OS Programmer's guide for further details.	Use a call statement in PL/I to call a Pascal procedure. The PL/I procedure should specify the Pascal as an EXTERNAL. After the last call to a Pascal procedure you must call PSCLHX.

Figure 79. Inter Language Communication

The details of Pascal/VS linkage conventions are discussed in the chapter "Code Generation for the IBM/370" on page 93. You should familiarize your-

self with this section - especially if you plan to use Assembler language.

13.1 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 97, 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 93.)

13.1.1 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 80, 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 80. Minimum interface to an Assembler routine: skeletal code to be invoked from Pascal/VS

13.1.2 Writing Assembler Routine with General Interface

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

         EPILOG  DROP=[YES]
                      [NO ]
```

where:

procname is the entry point name of the routine.

LASTREG is a number between 3 and 12, inclusive, which indicates the highest register to be modified by the routine between 3 and 12.

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.

DROP indicates whether register 2 is to be dropped as a base register after the epilogue is executed.

defaults:
LASTREG=12
VARS=0
PARMS=0

DROP=YES

Figure 81. 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 4096 bytes 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 4096 bytes 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 macro will cause register 2 to be dropped as a base register unless DROP=NO is specified.

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
         END
```

Figure 82. General interface to an Assembler routine: skeletal code to be invoked from Pascal/VS

13.1.3 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 97. At invocation register 1 contains the address of this list.

If the general interface (see "Writing Assembler Routine with General Interface" on page 107) is used in writing the Assembler routine, passed-in parameters start at offset 144 from register 13 after the prologue has been executed.

13.1.4 Calling Pascal/VS Routine from Assembler Routine

An Assembler language routine that was invoked from a Pascal program may call a Pascal procedure provided that:

- the general Pascal/VS interface was incorporated within the Assembler routine, and
- the Pascal/VS routine to be called is declared as external.

See Figure 84 on page 110 as an example.

If the Assembler routine was not invoked from a Pascal/VS routine, then the Pascal/VS run time environment must be set up prior to entering the Pascal/VS routine. To do this, the Pascal procedure must be declared with the MAIN or REENTRANT directive. (See Figure 86 on page 112 for an example.) When such a procedure is invoked for the first time, a minimum environment is created. On subsequent calls, this environment is restored prior to executing the procedure. To remove the environment (free stack space, etc.), the procedure PSCLHX is provided.

Prior to making the call to a Pascal procedure from Assembler language, 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 97.

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.

13.1.5 Sample Assembler Routine

In Figure 83 on page 110 and Figure 84 on page 110, 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 83. Pascal/VS description of Assembler routine: the Assembler routine is shown in Figure 84.

```
TIOSEG   CSECT
TPUT     PROLOG LASTREG=4,VARS=8  only registers 3 and 4 are modified
*
         L      3,144(13)         load address of 'BUF' parameter
         L      4,148(13)         load 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
*
TPUTMSG  DC     AL2(L'TPUTTEXT)   halfword length of string
TPUTTEXT DC     C'TPUT ERROR'     message text
         END
```

Figure 84. Sample Assembler routine: this routine is invoked by a Pascal/VS routine and, within itself, invokes a Pascal/VS routine.

13.1.6 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 85. Example of calling a Pascal/VS program from an Assembler routine

Pascal/VS procedure to be called:

```
SEGMENT SQUARE;
procedure SQUARE(var X : REAL);
    MAIN;
procedure SQUARE;
   begin
     X := X * X
   end; .
```

Assembler routine to call Pascal/VS procedure:

```
TOSQ     CSECT
         USING *,15                establish addressability
         STM   14,12,12(13)        save callers registers
         ST    13,SAVEAREA+4       save address of callers save area
         BALR  2,0
         USING *,2                 establish addressability
         LA    13,SAVEAREA         set new save area
         LA    1,PLIST1            Reg 1 points to parameter list
         L     15,=V(SQUARE)       load address of Pascal procedure
         BALR  14,15               call SQUARE
         LA    1,PLIST2            reg 1 points to parameter list
         L     15,=V(PSCLHX)       load address of Pascal procedure
         BALR  14,15               call PSCLHX to terminate environment
         L     13,SAVEAREA+4       return
         LM    14,12,12(13)
         BR    14
PLIST1   DC    A(X)                PARAMETER LIST
X        DC    D'4.0'
PLIST2   DC    A(ZERO)             PARAMETER LIST
ZERO     DC    F'0'
SAVEAREA DS    18F
         END
```

Figure 86. Example of Assembler as the caller to Pascal/VS

Pascal/VS program which invokes an Assembler routine named SUM:

```
     program FROMPSCL;                        (*Pascal program heading      *)
       procedure SUM(var I : INTEGER;
                     const J : INTEGER);
         FORTRAN;
     var
       I,J   :INTEGER;                          (*Define two local variables  *)
     begin
       I := 0;                                  (*Set running sum to zero      *)
       for J := 1 to 10 do                      (*loop through ten values      *)
         begin
           SUM(I,J);                            (*compute the next sum         *)
           WRITELN('The current running sum is ',I:0);
         end;
     end .                                      (*FROMPSCL                     *)
```

Assembler routine which is being invoked from Pascal program:

```
SUM       CSECT
          USING *,15                establish addressability
          STM   14,12,12(13)        save callers registers
          ST    13,SAVEAREA+4       save address of callers save area
          BALR  5,0
          USING *,5                 establish addressability
          LA    13,SAVEAREA         set new save area
          L     2,0(1)              get address of I
          L     3,0(2)              get I
          L     4,4(1)              get address of J
          A     3,0(4)              I = I + J
          ST    3,0(2)              return the new value of I
          L     13,SAVEAREA+4       return
          LM    14,12,12(13)
          BR    14
SAVEAREA  DS    18F
          END
```

Figure 87. Example of Pascal/VS as the caller to Assembler

13.2 PASCAL/VS AND FORTRAN

Communication between FORTRAN and Pascal/VS is accomplished by use of the MAIN directive (FORTRAN to Pascal/VS) and the FORTRAN directive (Pascal/VS to FORTRAN).

Data may be passed between FORTRAN and Pascal/VS through the parameter list or FORTRAN COMMON. If you choose to COMMON, specify the name of the COMMON block as a Pascal/VS **def** variable.

13.2.1 Pascal/VS as the Caller to FORTRAN

Pascal/VS program that calls a FORTRAN subroutine:

```
program FROMPSCL;                          (*Pascal program heading      *)
  procedure SUM(var I : INTEGER;
                const J : INTEGER);
    FORTRAN;
var
  I,J   :INTEGER;                          (*Define two local variables  *)
begin
  I := 0;                                  (*Set running sum to zero      *)
  for J := 1 to 10 do                      (*loop through ten values      *)
    begin
      SUM(I,J);                            (*compute the next sum         *)
      WRITELN('The current running sum is ',I:0);
    end;
end .                                      (*FROMPSCL                     *)
```

FORTRAN subroutine:

```
      SUBROUTINE SUM(I,J)
      I = I + J
      RETURN
      END
```

Figure 88. Example of Pascal/VS as the caller to FORTRAN

The FORTRAN directive instructs Pascal/VS to utilize exactly the same calling conventions employed by FORTRAN. This restricts the form of the parameter list, namely you may not pass a parameter by value; you may pass a parameter by **var** or by **const**. If you choose the latter mechanism, the FORTRAN subprogram must not modify the parameter.

Execution errors that occur during the execution of the FORTRAN program will be handled by the Pascal runtime support routines. If you desire to enable the error handling of FORTRAN you should invoke "VSCOM#" at the appropriate entry point. Consult the VS FORTRAN Application Programming Guide SC26-3985 for details

13.2.2 FORTRAN as the Caller to Pascal/VS

Pascal/VS procedure to be called from FORTRAN program:

```
      SEGMENT SQUARE;
      procedure SQUARE(var X : REAL);
            MAIN;
      procedure SQUARE;
         begin
           X := X * X
         end;.
```

FORTRAN program that invokes Pascal procedure:

```
      REAL*8 AREAL
      AREAL = 4.0
      CALL SQUARE(AREAL)
      PRINT 1, AREAL
      CALL SQUARE(AREAL)
      PRINT 1, AREAL
      CALL SQUARE(AREAL)
      PRINT 1, AREAL
      CALL SQUARE(AREAL)
      PRINT 1, AREAL
C        TERMINATE PASCAL ENVIRONMENT
      CALL PSCLHX(0)
      STOP
    1 FORMAT (F12.0)
      END
```

Figure 89. Example of FORTRAN as the caller to Pascal/VS

Pascal/VS permits a FORTRAN program to call a Pascal procedure as a subprogram. To do this you specify the Pascal procedure with the MAIN directive.

The first invocation of any procedure with a MAIN directive will cause Pascal to establish the appropriate environment for its execution. Subsequent calls will use the same environment that was set up on the first call.

It is your responsibility to clean up the Pascal environment; this is done by invoking the procedure "PSCLHX".

If Pascal is not the main program, then Pascal will not attempt to handle any errors during execution.

13.3 PASCAL/VS AND COBOL

Communication between COBOL and Pascal/VS is accomplished by use of the MAIN directive (COBOL to Pascal/VS) and the FORTRAN directive (Pascal/VS to COBOL).

13.3.1 Pascal/VS as the Caller to COBOL

Pascal program that calls a COBOL subprogram:

```
program FROMPSCL;                        (*Pascal program heading      *)
  procedure SUMX(var I : INTEGER;
                 const J : INTEGER);
    FORTRAN;
var
  I,J   :INTEGER;                          (*Define two local variables  *)
begin
  I := 0;                                  (*Set running sum to zero      *)
  for J := 1 to 10 do                      (*loop through ten values      *)
    begin
      SUMX(I,J);                           (*compute the next sum         *)
      WRITELN('The current running sum is ',I:1);
    end;
end .                                      (*FROMPSCL                     *)
```

COBOL subprogram:

```
      IDENTIFICATION DIVISION.
      PROGRAM-ID. SUMX.
      ENVIRONMENT DIVISION.
      CONFIGURATION SECTION.
      SOURCE-COMPUTER. IBM-370.
      OBJECT-COMPUTER. IBM-370.
      DATA DIVISION.
      LINKAGE SECTION.
      77 I PIC IS 999999999 USAGE IS COMPUTATIONAL.
      77 J PIC IS 999999999 USAGE IS COMPUTATIONAL.
      PROCEDURE DIVISION USING I J.
         ADD J TO I.
         GOBACK.
```

Figure 90. Example of Pascal/VS as the caller to COBOL

The FORTRAN directive instructs Pascal/VS to utilize exactly the same calling conventions employed by FORTRAN which is also equivalent to COBOL. This restricts the form of the parameter list, namely you may not pass a parameter by value; you may pass a parameter by **var** or by **const**. If you choose the latter mechanism, the COBOL subprogram must not modify the parameter.

Execution errors that occur during the execution of the COBOL program will be handled by the Pascal runtime support routines. Pascal will not issue a call to ILBOSTP0 (which sets up the COBOL error recovery). You may call this routine if you would like the "STOP RUN" statement of COBOL to treat the Pascal calling procedure as a main entry point of a COBOL program. Consult the OS/VS COBOL Compiler and Library Programmer's Guide, SC28-6483 for details.

A COBOL program which is communicating with Pascal/VS must not use the dynamic loading feature.

13.3.2 COBOL as the Caller to Pascal/VS

Pascal procedure that is to be called from COBOL:

```
SEGMENT SQUARE;
procedure SQUARE(var X : REAL);
      MAIN;
procedure SQUARE;
   begin
     X := X * X
   end; .
```

COBOL program which calls a Pascal procedure:

```
    IDENTIFICATION DIVISION.
    PROGRAM-ID. TOSQ.
    ENVIRONMENT DIVISION.
    CONFIGURATION SECTION.
    SOURCE-COMPUTER. IBM-370.
    OBJECT-COMPUTER. IBM-370.
    DATA DIVISION.
    LINKAGE SECTION.
    77     AREAL USAGE IS COMPUTATIONAL-2.
    77     AZERO USAGE IS COMPUTATIONAL PIC IS 999999999.
    PROCEDURE DIVISION.
        MOVE 2 TO AREAL.
        CALL "SQUARE" USING AREAL.
        DISPLAY AREAL.
        MOVE 0 TO AZERO.
        CALL "PSCLHX" USING AZERO.
        MOVE 0 TO RETURN-CODE.
        STOP RUN.
```

Figure 91. Example of COBOL as the caller to Pascal/VS

Pascal/VS permits a COBOL program to call a Pascal procedure as a subprogram. To do this you specify the Pascal procedure with the MAIN directive.

The first invocation of any procedure with a MAIN directive will cause Pascal to establish the appropriate environment for its execution. Subsequent calls will use the same environment that was created in the first call.

It is your responsibility to clean up the Pascal environment, this is done by invoking the procedure "PSCLHX". If Pascal is not the main program, then Pascal will not attempt to handle any errors during execution.

13.4 PASCAL/VS AND PL/I

Communication between PL/I and Pascal/VS is accomplished by use of the MAIN directive (PL/I to Pascal/VS) and the FORTRAN directive (Pascal/VS to PL/I). In addition, you may use the REENTRANT directive instead of the MAIN directive in order to develop a REENTRANT call to Pascal.

13.4.1 Pascal/VS as the Caller to PL/I

Pascal program which calls a PL/I procedure:

```
program FROMPSCL;                        (*Pascal program heading      *)
  procedure SUM(var I : INTEGER;
                const J : INTEGER);
    FORTRAN;
var
  I,J   :INTEGER;                          (*Define two local variables  *)
begin
  I := 0;                                  (*Set running sum to zero      *)
  for J := 1 to 10 do                      (*loop through ten values      *)
    begin
      SUM(I,J);                            (*compute the next sum         *)
      WRITELN('The current running sum is ',I:0);
    end;
end .                                      (*FROMPSCL                     *)
```

PL/I procedure that is invoked from Pascal:

```
 SUM: PROC (I,J) OPTIONS(FORTRAN);
      DCL (I,J) FIXED BINARY(31,0);
      I = I + J;
      RETURN;
      END;
```

Figure 92. Example of Pascal/VS as the caller to PL/I

The FORTRAN directive instructs Pascal/VS to utilize exactly the same calling conventions employed by FORTRAN. PL/I will employ FORTRAN calling conventions if "FORTRAN" is specified in the OPTIONS clause. Consult the PL/I Programmer's Guide, SC33-0037 (CMS) and SC33-0006 (OS) for details.

The FORTRAN directive restricts the form of the parameter list, namely you may not pass a parameter by value; you may pass a parameter by either **var** or **const**. If you choose to latter mechanism, the PL/1 procedure must not modify the parameter.

13.4.2 PL/I as the Caller to Pascal/VS

Pascal procedure which is called from PL/I:

```
     SEGMENT SQUARE;
     procedure SQUARE(var X : REAL);
          MAIN;
     procedure SQUARE;
        begin
          X := X * X
        end; .
```

PL/I program which calls a Pascal procedure:

```
 TOSQ: PROC OPTIONS(MAIN);
       DCL SQUARE ENTRY EXTERNAL;
       DCL PSCLHX ENTRY(FIXED BINARY(31,0)) EXTERNAL;
       DCL ZERO FIXED BINARY(31,0);
       AREAL = 4.0;
       CALL SQUARE(AREAL);
       PUT LIST(AREAL);
       CALL SQUARE(AREAL);
       PUT LIST(AREAL);
       CALL SQUARE(AREAL);
       PUT LIST(AREAL);
       CALL SQUARE(AREAL);
       PUT LIST(AREAL);
       ZERO = 0;
       CALL PSCLHX(ZERO);
       END;
```

Figure 93. Example of PL/I as the caller to Pascal/VS

Pascal procedure which is called from a reentrant PL/I program:

```
SEGMENT SQUARE;
procedure SQUARE(var E : INTEGER; var X : REAL);
      REENTRANT;
procedure SQUARE;
   begin
     X := X * X
   end; .
```

ReentraAt PL/I program which invokes a Pascal procedure:

```
TOSQ: PROC OPTIONS(MAIN REENTRANT);
      DCL SQUARE ENTRY EXTERNAL;
      DCL PSCLHX ENTRY(FIXED BINARY(31,0)) EXTERNAL;
      DCL SAVE FIXED BINARY(31,0);
      AREAL = 4.0;
      SAVE = 0;
      CALL SQUARE(SAVE,AREAL);
      PUT LIST(AREAL);
      CALL SQUARE(SAVE,AREAL);
      PUT LIST(AREAL);
      CALL SQUARE(SAVE,AREAL);
      PUT LIST(AREAL);
      CALL SQUARE(SAVE,AREAL);
      PUT LIST(AREAL);
      CALL PSCLHX(SAVE);
      END;
```

Figure 94. Example of PL/I as the caller to Pascal/VS: Use of the REENTRANT directive

Pascal/VS permits a PL/I program to call a Pascal procedure as a subprogram. To do this you specify the Pascal procedure with the MAIN directive.

The first invocation of any procedure that has a MAIN directive associated with it will cause Pascal to establish the appropriate environment for its execution. Subsequent calls will use the same environment that was created on the first call.

A call to PSCLHX will dispose of the Pascal environment and release all memory that it utilizes.

The Pascal/VS run time support will not attempt to handle any errors during execution, unless the main program is in Pascal.

The REENTRANT directive may be used in place of the MAIN directive if the program must be reentrant. In this case you must assist Pascal/VS in keeping track of the location of the Pascal/VS execution environment. The first parameter to a REENTRANT procedure must be an integer passed by **var**. The first call to the procedure must pass as its first parameter, a FIXED BIN(31,0) variable which has been set to the value zero. Upon return from the first call, this variable will contain an address which refers to the newly created Pascal/VS environment. This variable should be passed unaltered to subsequent calls so that the Pascal/VS environment may be reentered.

To terminate the Pascal/VS environment that was set up by the REENTRANT procedure, the "PSCLHX" should be called with the variable that contains the address. See Figure 94 for an example.

13.5 DATA TYPES COMPARISON

Every language has numerous data types that are suited for the applications for which the language was intended. When passing data between programs written in different languages you must be aware which data types are the same and where there is no equivalent representation.

Some data types in other languages have no direct equivalent in Pascal; however, you can often create new user data types in Pascal that will simulate some of the data types found in other languages. For example, you could define a record type that is identical to FORTRAN's COMPLEX type.

Figure 95 on page 121 compares Pascal data types with the equivalent in FORTRAN, COBOL and PL/I.

Pascal/VS makes no attempt to remap any storage when an inter-language call is made. This means that beause FORTRAN stores its arrays in column-major order and Pascal stores its arrays in row-major order, a call between FORTRAN and Pascal/VS procedures appears to transpose the array.

Data Type Equivalences Between Different Langauges			
Pascal/VS	FORTRAN	COBOL	PL/I
CHAR	CHARACTER*1	PIC X	CHAR
BOOLEAN	LOGICAL*1	na	FIXED BINARY(1,0)
INTEGER	INTEGER*4	PIC S999999999 USAGE IS COMP	FIXED BINARY(31,0)
packed -32768..32767	INTEGER*2	PIC S9999 USAGE IS COMPUTATIONAL	FIXED BINARY(15,0)
packed 0..65536	na	na	na
packed -128..127	na	na	FIXED BINARY(7,0)
packed 0..255	na	na	na
REAL	REAL*8	COMPUTATIONAL-2	REAL FLOAT DEC(16)
SHORTREAL	REAL*4	COMPUTATIONAL-1	REAL FLOAT DEC(6)
packed array[1..n] **of** CHAR	CHARACTER*n	PIC X(n) or PIC X OCCURS n TIMES	CHAR(n)
STRING(m)	na	na	CHAR(m) VARYING
set of 0..n	na	na	BIT(n+1)
@ id	na	na	POINTER
array	dimensioned variable	OCCURS	dimensioned variable
record	na	record	structure
space	na	na	AREA

Figure 95. Data Type Comparisons

14.1 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.

14.2 THE MAIN PROGRAM

The main program is called as an ordinary procedure from the environment setup routine (PASCALVS). The entry point name of the main program is AMPXBEGN.

14.3 EXECUTION SUPPORT ROUTINES

Execution Support Routines	
Procedure name	Action Performed
AMPXBCLK	Initializes the execution clock
AMPXCHKS	Checks a set for membership
AMPXCLCK	Interogate the execution clock
AMPXCRTE	Initialize the PCWA
AMPXDATE	DATETIME procedure
AMPXDATI	System date and time
AMPXDBCB	Obtains a procedures DBCB pointer
AMPXECLK	Ends the the execution clock
AMPXGOTO	Handles goto out of block
AMPXGTOK	Obtains a token from user's execution parameters
AMPXG12	Returns the contents of register 12
AMPXG13	Returns the contents of register 13
AMPXHALT	HALT procedure
AMPXINIT	Initializes prior to execution of a Pascal program
AMPXMAIN	Interface for calling Pascal for other languages
AMPXMOVE	Memory to memory move
AMPXMUS	Adds elements to a set
AMPXNAME	Obtains a procedures name
AMPXPAD	Memory fill memory with blanks
AMPXPARM	PARMS function
AMPXRETC	RETCODE procedure
AMPXSETV	Memory fill of with a value
AMPXSPAR	Intialize for PARMS function
AMPXTERM	Termination after execution of a Pascal program
AMPXTOK	TOKEN procedure
AMPXTRAC	TRACE procedure
AMPZABND	Abnormal termination routine
AMPZCVD	Convert to decimal
CMS	CMS procedure
PASCALVS	Main entry point for a Pascal/VS main program
PSCLHX	Terminates execution for interlanguage calls

These routines provide miscellanaous functions such as program initialization and low level routines such as fast memory move.

14.4 INPUT/OUTPUT ROUTINES

Internal Input/Output Routines	
Procedure name	Action Performed
AMPXCLOS	CLOSE procedure
AMPXCOLS	COLS function
AMPXGET	GET procedure (TEXT files)
AMPXGETR	GET procedure
AMPXOPEN	RESET, REWRITE or UPDATE procedures
AMPXOPN1	Initializes a PCB prior to opening
AMPXOPN2	Sets a PCB after opening
AMPXPARS	Analyze the optional string on RESET or REWRITE
AMPXPCBC	Closes a file (PCB)
AMPXPDS	PDS support routines (PDSIN and PDSOUT)
AMPXPUT	PUT procedure
AMPXRCHR	Reads into a CHAR
AMPXRINT	Reads into an INTEGER
AMPXRLIN	Reads to end of line (TEXT file)
AMPXRR	Reads a REAL value
AMPXRRDY	Prepares a TEXT file for input
AMPXRREC	Reads one record (non TEXT files)
AMPXRSTR	Reads into a STRING
AMPXRTXT	Reads into an array of CHAR
AMPXSEEK	SEEK procedure
AMPXSTAT	Obtains the status of a file
AMPXTIO	Terminate I/O processing
AMPXWB	Writes a BOOLEAN value
AMPXWCHR	Moves data to an I/O output buffer
AMPXWCHS	Writes a CHAR to a TEXT file
AMPXWINT	Writes an INTEGER to a TEXT file
AMPXWLIN	Writes an end-of-line to a TEXT file
AMPXWR	Writes a REAL value
AMPXWRDY	Prepares a TEXT file for output
AMPXWREC	Writes one record (non TEXT files)
AMPXWSTG	Writes a string to a TEXT file
AMPXWTXT	Writes an array of CHAR to a TEXT file
AMPYCLOS	System dependent QSAM close
AMPYDFLT	Applies System dependent defaults to a file
AMPYGET	System dependent get procedure
AMPYOPEN	System dependent QSAM open
AMPYPAGE	PAGE procedure
AMPYPDS	System dependent PDS interface
AMPYPUT	System dependent put procedure
AMPYSEEK	System dependent seek procedure
AMPZDAMR	Issues a READ for a BDAM data set
AMPZDAMW	BDAM write procedure
AMPZDCBC	Close on an OS DCB
AMPZDCBO	Open on an OS DCB
AMPZFIND	Issues OS FIND
AMPZGET	Issues a QSAM GET
AMPZPUT	Issues a QSAM PUT
AMPZPUTX	Issues a QSAM PUTX
AMPZSAMR	Issues a READ for a BSAM data set
AMPZSAMW	BSAM write procedure
AMPZSTOW	Issues OS STOW
AMPZTGET	Issues a TGET (OS) or RDTERM (CMS)
AMPZTPUT	Issues a TPUT (OS) or WRTERM (CMS)

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.

14.5 ERROR HANDLING

Error Handling	
Procedure name	Action Performed
AMPXCHKR	Intercepts execution time checking errors
AMPXDIAG	Intercepts program exceptions
AMPXERR	General execution time error handler
AMPXIOER	I/O error intercept routine
ONERROR	Default ONERROR procedure

When the runtime environment detects an error condition, it calls a routine to handle the error. There are several different routines, one routine for each of class of error (e.g. I/O error, program exception etc). The routine AMPXERR is the central routine, it is always called from the other routines: it then calls ONERROR, the user provided error handler, and then completes the error handling.

14.6 CONVERSION ROUTINES

Conversion Routines	
Procedure name	Action Performed
AMPTTOR	Converts a REAL (EBCDIC) to REAL
AMPXBTOS	BOOLEAN to string conversion
AMPXCTOS	Converts a CHAR to a string
AMPXSTOS	Converts a string to a string
AMPXITOS	Converts an INTEGER to a string
AMPXOTOS	Converts an offset in a procedure to a statement number
AMPXPACK	PACK procedure
AMPXRTOS	Conversion for a REAL to a STRING
AMPXSTOC	Conversion for a STRING to a CHAR
AMPXSTOG	Conversion for a STRING to a STRING
AMPXSTOI	Conversion for a STRING to an INTEGER
AMPXSTOR	Converts a REAL (EBCDIC) to REAL
AMPXSTOT	Conversion for a STRING to an array of CHAR
AMPXTTOS	Appends an array of CHAR to a string
AMPXUCAS	Lower case to upper case conversion
AMPXUNPK	UNPACK procedure
ITOHS	Integer to hexadecimal string conversion

There are several places where Pascal/VS must perform data conversions. They take place when you are doing I/O on TEXT files and when you use READSTR and WRITESTR.

14.7 MATHEMATICAL ROUTINES

Mathematical Routines	
Procedure name	Action Performed
AMPXATAN	ARCTAN function
AMPXCOS	COS function
AMPXEXP	EXP function
AMPXLN'	LN function
AMPXRAND	RANDOM procedure
AMPXSIN	SIN function
AMPXSQRT	SQRT

The predefined functions are provided as Pascal/VS functions. The Pascal/VS compiler changes the user provided name (e.g. SIN) to an internal name (e.g. AMPXSIN).

14.8 STRING ROUTINES

String Routines	
Procedure name	Action Performed
AMPX$COM	COMPRESS function (long strings)
AMPX$DEL	DELETE function (long strings)
AMPX$LTR	LTRIM procedure (long strings)
AMPX$SUB	SUBSTR function (long strings)
AMPX$TRI	TRIM function (long strings)
AMPXCAT	Concatenates 2 to 9 strings
AMPXCOMP	COMPRESS function (short strings)
AMPXDELE	DELETE function (short strings)
AMPXINDX	INDEX procedure
AMPXLTRI	LTRIM procedure (short strings)
AMPXSUBS	SUBSTR function (short strings)
AMPXTRIM	TRIM function (short strings)
LPAD	LPAD procedure (see Appendix C)
PICTURE	PICTURE function (see Appendix C)
RPAD	RPAD procedure (see Appendix C)

The predefined functions and procedures are provided as Pascal/VS functions and procedures. The Pascal/VS compiler changes the user provided name (e.g. SUBSTR) to an internal name (e.g. AMPXSUBS). Several routines are provided in two forms: long and short. The short form is always used if possible. In order to use the short form the Pascal/VS compiler must determine that the resulting string will be less than 1000 bytes long. If the size can't be limited by compiler analysis, the compiler uses the long form which passes the results through the heap.

14.9 MEMORY MANAGEMENT ROUTINES

Memory Management Routines	
Procedure name	Action Performed
AMPXALOC	Basic storage allocator
AMPXDISP	DISPOSE procedure
AMPXFREE	Basic storage de-allocator
AMPXIDSP	Dispose for the I/O routines
AMPXINEW	New for the I/O routines
AMPXMARK	MARK procedure
AMPXNEW	NEW procedure
AMPXVNEW	NEW procedure (when record is allocated with tags)
AMPXRLSE	RELEASE procedure
AMPXTMEM	Termination processing for memory management

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

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

The RELEASE procedure 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.

The I/O routines have access to a separate heap is controlled with the routines AMPXINEW and AMPXIDSP. Thus, I/O buffers and file control blocks are in a distinct area from the users area.

Release 2.1 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.

15.1 PASCAL/VS RESTRICTIONS

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

Conformant array parameters
The conformant array mechanism for passing array variables to routines is not supported.

Note: Conformant arrays are only required by the ISO level 1 standard; the ISO level 0 standard and the ANSI standard do not require them.

Note: In Release 2.0, procedures which are passed as parameters were restricted to the outer most nesting level. In Release 2.1, this restriction was removed.

15.2 MODIFIED FEATURES

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

15.3 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 maximum length of a STRING is 32767 characters.
- The STRING operators and functions are concatenate, LENGTH, STR, SUBSTR, DELETE, TRIM, LTRIM, COMPRESS, INDEX, TOKEN, READSTR and WRITESTR.
- A new predefined type, STRINGPTR, has been added that permits you to allocate strings with the NEW procedure whose maximum size is not defined until the invocation of NEW.
- A new parameter passing mechanism is provided that allows strings to be passed into a procedure or function without requiring you to specify the maximum size of the string on the formal parameter.
- The MAXLENGTH function returns the maximum length that a string variable can assume.
- Calls to FORTRAN subroutines and functions are provided for.
- The MAIN directive permits you to define a procedure that may be invoked from a non Pascal environment. A procedure that uses this directive is not reentrant.
- The REENTRANT directive permits you to define a procedure that may be invoked from a non Pascal environment. A procedure that uses this directive is reentrant.
- 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 optional string parameter on the procedures: RESET, REWRITE, UPDATE, TERMIN, TERMOUT, PDSIN and PDSOUT.
- The parameters of the text file READ procedure may be length-qualified.
- Files may be opened for updating with the UPDATE procedure.
- Input files may be opened as "INTERACTIVE" so that I/O may be done conveniently from a terminal.

Files may be opened for terminal input (TERMIN) and terminal output (TERMOUT) so that I/O may take place directly to the user's terminal without going through the DDname interface.

Files may be accessed based on relative record number (random access).

The PDSIN procedure opens a partitioned dataset (or MACLIB) for input. The PDSOUT procedure opens a partitioned dataset (or MACLIB) for output. A string parameter is required to set the member name.

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 binary and 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: DATETIME, CLOCK, PARMS and RETCODE.
- Constants may be of a structured type (namely arrays and records).
- To control the compiler listing, the following listing directives are supported: %PAGE, %CPAGE, %SKIP, and %TITLE.
- The HALT procedure has been added to exit the program from an arbitrary location.
- The TRACE procedure prints the trail of routine invocations.
- The LOWEST, HIGHEST, LBOUND, HBOUND and SIZEOF functions provide a means of acquiring information variables and types.
- MARK and RELEASE provide a means of controlling dynamic variable allocation.
- Both single and double precision floating point numbers are provided using the SHORTREAL and REAL types.
- Identifiers may contain a dollar sign ($) anywhere a letter may go and an underscore (_) anywhere a digit may go.
- The predefined constants MINREAL and MAXREAL contain the values of the smallest and largest real numbers, respectively.
- The ADDR function returns the address of a variable.

16.1 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
- VM/PC
- OS/VS2 TSO
- OS/VS2 Batch

16.2 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.

The compiler is reentrant and may be loaded in a shared area in MVS or mapped to a shared segment in CMS. However, the Pascal/VS PID materials do not contain procedures to do this.

16.3 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 Language Reference Manual for more details.

Floating-point
Some commonly required characteristics of System/370 floating-point arithmetic are shown in Figure 96 on page 132.

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.[17] 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, ADDR, CHR, CLOSE, DISPOSE, EOF, EOLN, FLOAT, GET, HBOUND, HIGHEST, LBOUND, LENGTH, LOWEST, MARK, MAX, NEW, ODD, ORD, PACK, PAGE, PDSIN, PDSOUT, PRED, PUT, READ, READLN, READSTR, RELEASE, RESET, REWRITE, ROUND, SIZEOF, SQR, STR, SUCC, TERMIN, TERMOUT,

[17] 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]	7.2370055773322 6E+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 Language Reference Manual.

[2] Maxreal is the largest finite floating-point number that may be represented. Its value is in the predefined constant MAXREAL.

[3] Minreal is the smallest positive finite floating-point number that may be represented. Its value is in the predefined constant MINREAL.

[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 96. Characteristics of System/370 floating point arithmetic

TRUNC, UNPACK, UPDATE, WRITE, WRITELN, WRITESTR

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

Size limitations

The size of a single procedure or function must not exceed 8192 bytes of generated code. 8192 bytes represent approximately 400 Pascal statements, depending on the complexity of the statements. The compiler will generate a diagnostic if this limit is reached.

17.1 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	**Routine may not be passed to FORTRAN subroutine** The indicated declaration of a FORTRAN subroutine (a procedure heading with the FORTRAN directive) contains an argument which is a procedure or function parameter. Procedures or functions may not be passed to FORTRAN subroutines.
10	**No case labels specified** A **case** statement with no case labels was found. A **case** statement may not be empty or consist only of an **otherwise** clause.
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 declared label** The indicated label has been previously declared 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	**The characters '$' and '_' are not valid in standard Pascal** This is a warning message that can occur when the LANGLVL(STANDARD) compile option is specified. An identifier is being declared which has a name containing characters which are not recognizable in "standard" Pascal.
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 or EXTERNAL must not specify any formal parameters. Any formal parameters are assumed to have been specified previously on the associated declaration that contained the FORWARD/EXTERNAL 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.
20	**String literal constant is too long: exceeds 3190** Because of an implementation restriction, a string constant may not exceed 3190 characters in length.
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 occurs when too many lexical scopes are active. This can occur in multiply nested **with** statements and **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 or EXTERNAL directive was already previously forwarded.
28	**Additional errors in this line were not diagnosed** The indicated construct contained more errors, but were not diagnosed 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 -2147483647 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, PDSIN, PDSOUT, READSTR, TERMIN, TERMOUT, UPDATE, WRITESTR
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 - SHORTREAL - CHAR - BOOLEAN - STRING - **packed array**[1..n] **of** CHAR where n is a positive integer constant.
44	**Field length 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 field 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.
47	**Array reference contains too many subscripts** An array variable of dimension 'n' is being subscripted with more than 'n' number of subscripts.
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.
50	**No max length specified on STRING type - 255 assumed** A type definition of the form "STRING" does not contain a length specification to indicate the maximum length of the string variable. 255 is the default length.
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	**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 pointer 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**
75	**Length qualifier applicable only to STRING type** A length qualifier was applied to a non-STRING type. STRINGs are the only types that may have length qualifiers.
76	**File of files not supported**

77	**Illegal reference of 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	**TEXT files may not be updated** An attempt was made to open a text file for updating. Only record files may be updated.
90	**Label not declared** The indicated label did not appear in a **label** declaration.

91	**Max length of string variable does not match formal parameter**
	
	A string variable is being passed to a procedure "by var" and the corresponding formal parameter is declared with an explicit length. This error occurs when the declared length of the variable being passed does not match that of the formal parameter.
	
	Example:
	
	`procedure XYZ(var S: STRING(100)); EXTERNAL;`
	`var T: STRING(50);`
	`begin`
	`  ...`
	`  XYZ(T);  (*ERROR: declared length of T does *)`
	`           (*   not match that of parameter S *)`
	`  ...`
	`end`

92	**"THEN" expected**
93	**Redundant case alternative** The indicated **case** statement label is equal to a previous label within the same **case** statement.
94	**Required length expression missing for dynamic string allocation** A pointer variable declared with the type STRINGPTR is being allocated with the NEW procedure, but the required length expression is missing.
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"**
118	**Magnitude of floating point constant too large or too small** The indicated floating point constant has a magnitude that is outside the range of the IBM/370 double precision representation. The largest floating point magnitude that can be represented is 7.2370055773322 6E75 The smallest is 5.39760534693403E-79
119	**First parameter of READSTR/WRITESTR must be of type STRING**
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.
121	**Declaration out of order: LABEL,CONST,TYPE,VAR,routine** This is a warning message that may be produced when the LANGLVL(STANDARD) compiler option is specified. One or more declaration constructs are not in the order required by standard Pascal. Standard Pascal requires identifiers to be declared in the following order: Labels Constants (**const**) Types (**type**) Variables (**var**) Routines (**procedure/function**)
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 32767 characters in length. 32767 is the maximum allowed length for a varying length string.
124	**Construct or operation is not in standard Pascal** This is a warning message that may be produced when the LANGLVL(STANDARD) compiler option is specified. The indicated language construct or arithmetic operation is not supported in "standard" Pascal, but is a Pascal/VS language extension.
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.
134	**Notify compiler maintenance group** If this error should occur, then notify the Pascal/VS compiler maintenance group at IBM. This is a compiler error.
135	**Notify compiler maintenance group** If this error should occur, then notify the Pascal/VS compiler maintenance group at IBM. This is a compiler error.
136	**Notify compiler maintenance group** If this error should occur, then notify the Pascal/VS compiler maintenance group at IBM. This is a compiler error.
137	**Passing packed record field by VAR not valid** This is a warning message that may be produced when the LANGLVL(STANDARD) compiler option is specified. The indicated field of a packed record is being passed as an actual **var** parameter to a routine. Passing fields of packed records as **var** parameters is not valid in standard Pascal.
138	**Passing SPACE component by VAR not valid** This is a warning message that may be produced when the LANGLVL(STANDARD) compiler option is specified. Standard Pascal requires that actual **var** parameters be properly aligned which is not necessarily the case with a **space** component. The indicated parameter is a component of a **space** variable which is being passed as a **var** parameter.
139	**Passing packed array element by VAR not valid** This is a warning message that may be produced when the LANGLVL(STANDARD) compiler option is specified. The indicated subscripted variable is being passed as an actual **var** parameter to a routine. The variable being subscripted is a packed array. Passing elements of packed arrays as **var** parameters is not valid in standard Pascal.

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.
141	**Symbol not recognizable in standard Pascal** This is a warning message that may result when the LANGLVL(STANDARD) compiler option is specified. The indicated symbol (or operator) is not supported in "standard" Pascal. The symbol is part of a construct which is a Pascal/VS language extension.
142	**Variable 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 consistent 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 either causes an overlap with a previous field within the record or is an illegal (negative) offset.
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 which is to be called from another module is nested within another routine which is not permitted. Such routines must be declared at the outermost nesting level.
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	**Notify compiler maintenance group** If this error should occur, then notify the Pascal/VS compiler maintenance group at IBM. This is a compiler error.

159	**Recursive type reference is not permitted**
	
	The compiler detected a degenerate **type** declaration of one of the following forms:
	I. `type X = X;`
	II. `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** A **else** symbol was detected that is not part of an **if** statement. This error often occurs when the preceding **then** clause of an **if** statement is terminated with a semicolon (;).
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 - MAIN - REENTRANT
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.
185	**Routine is not defined in standard Pascal** This warning may be produced when the LANGLVL(STANDARD) compiler option is specified. The indicated call statement refers to a predefined Pascal/VS routine which does not exist in standard Pascal.
186	**Directive only applies to procedure, not to a function** The indicated procedure directive ("MAIN" or "REENTRANT") is being applied to a function declaration. The directive is not supported for functions.
187	**Notify compiler maintenance group** If this error should occur, then notify the Pascal/VS compiler maintenance group at IBM. This is a compiler error.
188	**First parameter of REENTRANT procedure must be an integer by var** The indicated procedure declaration in which the directive "REENTRANT" was specified, failed to comply with the parameter list requirement for such a procedure: the first parameter of a "REENTRANT" procedure must be a pass-by-reference (specified with the **var** reserved word) integer in which a pointer to the Pascal/VS environment is saved between calls.
189	**Notify compiler maintenance group** If this error should occur, then notify the Pascal/VS compiler maintenance group at IBM. This is a compiler error.

190	**Notify compiler maintenance group** If this error should occur, then notify the Pascal/VS compiler maintenance group at IBM. This is a compiler error.
191	**Simple constant required** A constant expression which required compile-time computation was found where a simple constant is required. This is often a warning message that may be produced when the LANGLVL(STANDARD) compiler option is specified.
192	**%Percent directives are not recognized in standard Pascal** This warning may be produced when the LANGLVL(STANDARD) compiler option is specified. All compiler directives which appear in the source program with the percent (%) prefix are Pascal/VS extensions and are not supported in standard Pascal.

193	**FOR- or WHILE-loop has no statements within its body**
	
	This is a warning message to indicate that a **for**-statement or **while**-statement loops on an empty statement. Such a case is often not the programmer's intent.
	
	Examples
	
	`while A > 0 do;`
	
	`for I := 1 to J do ;`

194	**PACKED subranges not supported in standard Pascal** This warning may be produced when the LANGLVL(STANDARD) compiler option is specified. Subrange type definitions may not be "packed" in standard Pascal. This feature is a Pascal/VS language extension.
195	**Variable is not properly aligned** The indicated variable is being passed as a **var** parameter and the compiler has detected that its address may not be properly aligned. (For example, passing a full word integer which has an address that is not on a word boundary.) On most models of the 370 series, the manipulation of objects which are not properly align will result in a penalty in execution speed. This warning will be produced even if the variable is just potentially misaligned (as in the case of a subscripted variable).
196	**Offset qualification value is too large** The indicated field in a record definition is qualified with an offset which would result in a record that was too large too address.
197	**Object exceeds storage limits** The specified object would cause the program to require more storage than is physically addressable.

500	**Recursion detected in "%INCLUDE" processing lib(mem)**
	
	Source text which was included from member "mem" in library "lib" by means of the a %INCLUDE directive contains in itself a %INCLUDE directive which directly or indirectly references the same member recursively. This error causes immediate termination of the compilation.
	
	Example:
	
	Source program: Member TYPES:
	
	`program EXAMPLE;` `REC = record`
	`type` `NAME: STRING(10);`
	`%include TYPES;` `AGE : 0..99;`
	`begin` `end`
	`...` `%include TYPES; (*<===ERROR 500*)`
	`end.`

501	**Too many nesting levels in "%INCLUDE" processing lib(mem)** A %INCLUDE directive was detected which is nested 8 levels deep within a stack of "includes." "Included" source text may not be nested beyond 8 levels. This error causes immediate termination of the compilation.
502	**Unable to open "%INCLUDE" library: libname** The include library named "libname" could not be opened. Possible causes are that the DDname was not assigned or the DCB attributes of the library are not correct. This error causes immediate termination of the compilation.

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.

No.	Message and Explanation
AMP0001S	**Routine "name" is too large to compile at stmt n** The indicated routine has too many statements to compile; a fixed-length table of the compiler has overflowed. The last statement that was successfully processed was statement "n." The routine should be divided into two or more separate routines.

AMPT001E	**Inevitable NIL pointer error will occur**
	
	The code optimizer of the compiler has determined that a nil pointer checking error will inevitably occur at execution time at the specified routine and statement. Example:
	
	`begin`
	`  P := nil`
	`  WRITELN(P@.I);  (*<===AMPT001E - inevitable error*)`
	`end;`

AMP0002S	**Notify Pascal/VS Support - Optimizer error** An optimizer error occured at statement "nnn" of routine "xxxxxxxx" in module "mmmmmmmm". A message will be produced describing the kind of error. Notify Pascal/VS support.

AMPT002E	**Inevitable high bound error will occur**
	
	The code optimizer of the compiler has determined that a high bound checking error will inevitably occur at execution time at the specified routine and statement. Example:
	
	`var I : 1..10;`
	`  J : INTEGER;`
	`begin`
	`  J := 11;`
	`  I := J;  (*<===AMPT002E - inevitable error*)`
	`end;`

AMPT003E	**Inevitable low bound error will occur**
	
	The code optimizer of the compiler has determined that a low bound checking error will inevitably occur at execution time at the specified routine and statement. Example:
	
	`var I : 1..10;`
	`  J : INTEGER;`
	`begin`
	`  J := 0;`
	`  I := J;  (*<===AMPT003E - inevitable error*)`
	`end;`

AMPT005E	**Function routine does not return a value**
	
	The code optimizer of the compiler has determined that the specified function routine does not return a result. Example:
	
	`function F(var I: INTEGER): INTEGER;`
	`begin`
	`  READLN(I);`
	`end;  (*<===AMPT005 function did not return a result*)`

AMPT006E	**Expression is too complicated at stmt nnn of routine xxxxxxxx** The expression in statement "nnn" of routine "xxxxxxxx" is too complex to compile and should be broken up into multiple statements. If the indicated statement contains a relatively simple expression, then the Pascal/VS support group should be notified.
AMPT700S	**Routine "name" contains too many statements. Max=n** The statement table being generated overflowed in the specified routine. The routine should be divided into two or more routines.
AMPT701I	**Record type contains too many fields** The DEBUG compiler option was specified and a record type definition was compiled that contains too many fields to be accommodated in the debugger type table. If this error should occur, the resulting code may not work properly when the interactive debugger is enabled.
AMPT702S	**Routine "name" exceeds 8K limit at stmt n** The specified routine caused more than 8192 bytes of code to be generated starting at statement number "n." Since Pascal/VS only reserves two base registers to address code, 8192 bytes is the limit. The indicated routine should be divided into two or more separate routines.
AMPT703I	**Field name space pool overflowed** The DEBUG compiler option was specified and a large number of record type definitions were compiled. The debugger table which contains the record field names overflowed. If this error should occur, the resulting code may not work properly when the interactive debugger is enabled.
AMPT704I	**Type table overflow. Debug is disabled** The module being compiled with the DEBUG option contains more than 256 unique data types. The type table being generated for the interactive debugger may contain no more than 256 entries. The interactive debugger may not be used on this module.
AMPT705I	**Symbol name space pool overflowed** The DEBUG compiler option was specified and a large number of symbols were compiled. The debugger table which contains symbol names overflowed. If this error should occur, the resulting code may not work properly when the interactive debugger is enabled.
AMPL999S	**Compiler error notify Pascal/VS Support** An error was detected in the first pass of the compiler. If this error should occur, please notify Pascal/VS support at IBM.
AMPO999S	**Notify Pascal/VS Support - Optimizer Error** An error was detected in the second pass of the compiler. If this error should occur, please notify Pascal/VS support at IBM.
AMPT999S	**Notify Pascal/VS Support - Translation error** An error was detected in the third pass of the compiler. If this error should occur, please notify Pascal/VS support at IBM.

No.	Message and Explanation
AMPX011E	**Operation exception** An operation exception occurred in the program. The error is probably in an assembly language routine linked with your Pascal program or due to a 'wild' assignment through an uninitialized pointer.
AMPX012E	**Privileged exception** A privileged exception occurred in the program. The error is probably in an assembly language routine linked with your Pascal program.
AMPX013E	**Execute exception** An execute exception occurred in the program. The error is probably in an assembly language routine linked with your Pascal program.
AMPX014E	**Protection exception** A protection exception occurred in the program. The error is probably due to a 'wild' assignment through an uninitialized pointer, or to an array assignment with a bad subscript (with checking off).
AMPX015E	**Addressing exception** An addressing exception occurred in the program. The error is probably due to a 'wild' assignment through an uninitialized pointer, or to an array assignment with a bad subscript (with checking off).
AMPX016E	**Specification exception** A specification exception occurred in the program. The error is probably in an assembly language routine linked with your Pascal program.
AMPX017E	**Data exception** A data exception occurred in the program. The error is probably in a non-Pascal routine linked with a Pascal program.
AMPX018E	**Fixed point overflow exception** A fixed-point overflow exception occurred in the program. The error is probably due to an addition, subtraction, or multiplication that resulted in an integer with a magnitude which exceeds MAXINT.
AMPX019E	**Fixed point divide by zero exception** A fixed point divide by zero exception occurred in the program. The error is due to a **div** operation in which the second operand (the divisor) has the value zero.
AMPX020E	**Decimal overflow exception** A decimal overflow exception occurred in the program. The error is probably occurred in a non-Pascal routine linked to the Pascal program.

AMPX021E	**Decimal divide by zero exception** A decimal divide by zero exception occurred in the program. The error probably occurred in a non-Pascal routine linked to the Pascal program.
AMPX022E	**Exponent overflow exception** An exponent overflow exception occurred in the program. The error is probably due to a floating point multiplication or division which produces a result with a magnitude greater than 7.2370055773322 6E75.
AMPX023E	**Exponent underflow exception** An exponent underflow exception occurred in the program. The error is probably due to a floating point multiplication or division which produces a result with a magnitude less than 5.39760534693403E-79.
AMPX024E	**Significance exception** This exception is not intercepted by the Pascal/VS run time environment. If it should occur, then the Pascal/VS run time environment may have been locally modified. Contact your local system support.
AMPX025E	**Floating point divide by zero exception** A floating point divide by zero exception occurred in the program. The error is caused by an attempt to divide by zero.
AMPX026E	**Segment translation exception** This is a system error, run the program again and if the error persists contact Pascal/VS Development for assistance.
AMPX027E	**Page translation exception** This is a system error, run the program again and if the error persists contact Pascal/VS Development for assistance.
AMPX028E	**Translation specification exception** This is a system error, run the program again and if the error persists contact Pascal/VS Development for assistance.
AMPX029E	**Special operation exception** This is a system error, run the program again and if the error persists contact Pascal/VS Development for assistance.
AMPX030E	**Terminal attention exception** An attention was signaled from the users terminal.
AMPX031E	**Low bound checking error** Either the value of an array subscript, or the value being assigned to a subrange type variable is less than the minimum allowed for the subscript or subrange. This error may also result if the **mod** operation is attempted for which the second operand (the divisor) is less than or equal to zero.
AMPX032E	**High bound checking error** Either the value of an array subscript, or the value being assigned to a subrange type variable is greater than the maximum allowed for the subscript or subrange.

AMPX033E	**Nil pointer checking error** An attempt was made to reference a dynamic variable from a pointer which has the value **nil**.
AMPX034E	**Case label checking error** The expression of a **case**-statement has a value other than any of the specified **case** labels and there is no **otherwise** clause.
AMPX035E	**Function value checking error** A function routine returned to its invoker without being assigned a result.
AMPX036E	**Assertion failure checking error** The expression of an **assert** statement computed to the value FALSE.
AMPX037E	**String subscript out of bounds checking error** The subscript on a STRING was not in the range 0..LENGTH(s), where s is the STRING being subscripted.
AMPX038E	**Error 38 not assigned** This error number has not been assigned a meaning.
AMPX039E	**String truncation checking error** An assignment into a STRING variable could not be performed because the length of the source string is longer than the maximum length of the destination string.
AMPX040S	**Notify compiler maintenance group** If this error should occur, then notify the Pascal/VS compiler maintenance group at IBM. This is a run-time environment error.
AMPX041S	**File could not be opened: DDNAME** An error occurred when an attempt was made to open the file with the indicated DDname. The most probable cause of this error is a missing DDname definition. Under CMS, this error will occur when attempting to open a file that does not have a record format of 'F' or 'V'.
AMPX042E	**Lrecl size too small for file DDNAME** The logical record length of the file with the indicated DDNAME is not large enough to contain a single file component.
AMPX043E	**File is not open for output: DDNAME** An output operation was attempted on a file open for input.
AMPX044E	**File is not open for input: DDNAME** An input operation was attempted on a file open for output.
AMPX045E	**Logical record is too small in input file** A record file is being read which consists of varying length records (RECFM=V); and a logical record was read which is too short to represent a valid record in the file.
AMPX046E	**Data larger than lrecl for file** The logical record length of a file is too small to contain the file's component.

AMPX047E	**Invalid Input/Output option: xxxxx...** The options string passed to the procedure contains an incorrect or invalid option.
AMPX048E	**Missing member in file: member library** The indicated member could not be found in the partitioned data set.
AMPX049E	**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.
AMPX050E	**BLKSIZE exceeds 32760 in file DDNAME** A block size was specified that exceeds 32760 which is the maximum length of a block.
AMPX051E	**LRECL > BLKSIZE-4 in V format file: DDNAME** The logical record size was too large to permit at least one record to be fit in a block.
AMPX052E	**BLKSIZE not integer multiple of LRECL in DDNAME** The specified block size for a fixed-length record file is not an integer multiple of logical records.
AMPX053E	**Component length of file exceeds 32760 in DDNAME** A single element must fit in one logical record, therefore its length is restricted to 32760 bytes.
AMPX054E	**GET or READ called after end-of-file in DDNAME** An attempt was made to advance the file beyond the end-of-file.
AMPX055E	**Integer READ operation failed for file DDNAME** An attempt was made to read an integer from a text file, but either the end-of-file occurred, or an unrecognizable character was detected where the integer should have been.
AMPX056E	**Overflow/underflow detected in integer READ: DDNAME** An attempt was made to read an integer which has a value that does not lie within the range -2147483648..2147483647.
AMPX057E	**Invalid run time option:** An invalid option was specified when invoking a Pascal/VS program. A runtime option is specified preceding a slash '/' when invoking the program.
AMPX058I	**OPEN and INTERACTIVE are no longer supported, use READ/WRITE** The procedures OPEN and INTERACTIVE are not supported in Release 2.0 and up. The Pascal/VS Programmer's Guide SH20-6162-1 and the Pascal/VS Reference Manual SH20-6168-1 describes the equivalent operations.

AMPX059E	**Text exceeds logical record length in file "name"** A line of data is being written to the text file whose DDname is "name" and the line exceeded the logical record length of the file. As a recovery, the line is terminated at the end of the logical record and the remaining text of the line is placed in the next logical record. For each file being written, this error will be diagnosed only on the first occurrence; subsequent violations will not be diagnosed.
AMPX060E	**Operand to RELEASE does not correspond to MARK** The parameter passed to RELEASE did not have the value returned by a call to MARK.
AMPX061E	**Operand to 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.
AMPX062E	**Real READ operation failed for file DDNAME** An attempt was made to read an real from a text file, but either the end-of-file occurred, or an unrecognizable character was detected where the real should have been.
AMPX063E	**Operand to DISPOSE already deallocated** An attempt was made to perform a DISPOSE operation on a pointer which referenced heap storage which had been previously released.
AMPX064E	**Insufficient space to do NEW** There was not enough storage available to perform the NEW procedure. You should execute the program in a larger region (OS) or in a larger virtual machine (CMS). Also, you may not be calling DISPOSE for storage you no longer need.
AMPX065E	**Storage has been incorrectly assigned prior to DISPOSE** The pointer being disposed of was used incorrectly; namely, the pointer caused the heap to be modified beyond the size of the dynamic variable. This could happen if the dynamic variable was a **record** that was allocated by specifying tag values, and then was later used in an assignment with a different variant.
AMPX066E	**Operand to DISPOSE is NIL or undefined.** The operand is not valid for DISPOSE.
AMPX067E	**Heap incorrect due to previous invalid assignment using a pointer** The heap has been damaged. The cause of the damage was probably due to a pointer being used incorrectly.
AMPX068S	**Notify compiler maintenance group** If this error should occur, then notify the Pascal/VS compiler maintenance group at IBM. This is a run-time environment error.
AMPX069S	**Notify compiler maintenance group** If this error should occur, then notify the Pascal/VS compiler maintenance group at IBM. This is a run-time environment error.

AMPX070E	**LN: argument <= 0.0** The natural logarithm function (LN) was called with a 0 or negative argument.
AMPX071E	**SQRT: argument < 0.0, zero returned as result** The square root function (SQRT) was called with a negative argument.
AMPX072E	**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.
AMPX073E	**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).
AMPX074E	**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.
AMPX075E	**SEEK called for a file not opened for DIRECT access**
AMPX076E	**SEEK: bad relative record address** The record number in an invocation of SEEK has an invalid value.
AMPX077E	**Direct access file does not have fixed unblocked records: DDNAME** An attempt was made to perform direct access (relative record) operations on a file that was either not fixed or not unblocked. The required record format for a file to be manipulated with SEEK is RECFM=F.
AMPX078E	**Target string filled to maximum length in WRITESTR call** The target STRING (first parameter) in a call to WRITESTR was filled to capacity before the data being assigned into the STRING was exhausted.
AMPX079E	**Source string exhausted in READSTR call** Prior to reading all data from the the source string (first parameter), the end of the string was encountered.
AMPX080S	**Notify compiler maintenance group** If this error should occur, then notify the Pascal/VS compiler maintenance group at IBM. This is a run-time environment error.
AMPX081E	**LPAD: PADDING exceeds maximum length of string** The specified pad length (second operand) exceeds the maximum allowed length of the target string (first parameter).
AMPX082E	**DELETE: Length parameter less than zero**

AMPX083E	DELETE: starting index is less than 1
AMPX084E	DELETE: substring not contained within source string
AMPX085E	Set operation out of bounds An attempt to perform a set operation in which the resulting set contained members which are outside the range of a target set. This can occur in a set assignment in which the source set contains members which are not valid for the declared type of the target set.
AMPX086E	SUBSTR: Length parameter less than zero
AMPX087E	SUBSTR: starting index is less than 1
AMPX088E	SUBSTR: substring not contained within source string
AMPX089E	RPAD: padding exceeds maximum length of string The specified pad length (second operand) exceeds the maximum allowed length of the target string (first parameter).
AMPX200I	The module must be linked with DEBUG for debugger features An attempt was made to invoke the interactive debugger on a module that was not linked with the debugger library.
AMPX201I	The module must be linked with DEBUG for symbolic dump An execution time error occurred and a symbolic dump of the offending routine was attempted, but the module in which the routine is located was not compiled with the DEBUG option.
AMPX203I	Error occurred while executing ONERROR routine An execution time error has occurred while ONERROR was executing. ONERROR is a user provided procedure to diagnose execution errors and determine the correct course of action.
AMPX999S	NOTIFY PASCAL/VS SUPPORT: RECURSIVE ERROR IN RUNTIME ENVIRONMENT A second error was encountered while Pascal/VS was recovering from the first error. The program is terminated because any further processing would probably result in a CPU bound loop. You should notify Pascal/VS Development if this error persists.

17.3 MESSAGES FROM DEBUG

No.	Message and Explanation
AMPD500	Current module not compiled with Debug option
AMPD501	No statement *** in
AMPD502	There is no routine named * in module
AMPD503	Invalid qualification specification:
AMPD504	Missing qualification specification
AMPD505	Module name must be specified
AMPD506	Breakpoint is already set
AMPD507	Maximum number of breakpoints have been set
AMPD508	Specified breakpoint does not exist
AMPD509	is an automatic variable local to a non-active routine
AMPD510	Field qualified variable is not a record
AMPD511	is not a valid record field
AMPD512	Subscripted variable is not an array
AMPD513	Array subscript is not a scalar
AMPD514	Invalid symbol:
AMPD515	Array subscript is out of bounds:
AMPD516	Missing symbol:
AMPD517	Associated variable is not a pointer
AMPD518	Pointer variable does not contain valid address
AMPD519	not found in symbol table
AMPD520	Equate substitution is in infinite recursion
AMPD521	EQUATE expansion causes command truncation(exceeds 255 characters

AMPD522	You are not in CMS, command not valid
AMPD523	Debug command not recognized:
AMPD524	Invalid character in hexadecimal string:
AMPD525	Invalid hexadecimal string
AMPD526	Routine is not active
AMPD527	Qualification set to module
AMPD528	The word "EQUATE" may not be redefined
AMPD529	Maximum number of EQUATE''s have been set
AMPD530	There are no EQUATE''s currently set
AMPD531	Statement table missing Trace requires GOSTMT option
AMPD533	There are no active variables
AMPD534	Routine is not active:

17.4 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.

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.

APPENDIX A. 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 12 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: **SAMPLE.PASCAL** A partitioned data set containing sample programs.

File 11: **PASCALVS.MESSAGES** A sequential data set which contains the compiler messages.

File 12: CMS dump of the entire Pascal/VS package:

- **PASCALVS CONTENTS** A listing of the contents of the Pascal/VS package.
- **PASCALS MODULE** A program that issues all necessary FILEDEF commands to CMS prior to invoking the compiler.
- **PASCALL MODULE** The first pass of the compiler.
- **PASCALO MODULE** The second pass of the compiler.
- **PASCALT MODULE** The third pass of the compiler.
- **PASCALL TXTLIB** the txtlib from which PASCALL MODULE was generated.
- **PASCALO TXTLIB** the txtlib from which PASCALO MODULE was generated.
- **PASCALT TXTLIB** the txtlib from which PASCALT MODULE was generated.
- **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
- **PASCALVS CMSHELP** Help file that is accessed when "PASCALVS ?" is invoked.
- **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.
- **PRIMGEN PASCAL** A sample program.

File 13: **PASCALL.PASCAL** The source of the first pass of the compiler.

File 14: **PASCALO.PASCAL** The source of the second pass of the compiler.

File 15: PASCALT.PASCAL The source of the third pass of the compiler.

File 16: PASCALD.PASCAL The source of the interactive debugger.

File 17: PASCALX.PASCAL The source of the runtime library routines.

File 18: PASCALX.ASM The source of the operating system interface routines.

File 19: MACLIBL.PASCAL Include library for first pass of the compiler.

File 20: MACLIBO.PASCAL Include library for second pass of the compiler.

File 21: MACLIBT.PASCAL Include library for third 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 2300 blocks of free storage[18].

3. Access this disk with the ACCESS command.

4. Execute the following two commands:

```
TAPE FSF 11
TAPE LOAD * * m
```

where "m" is the single letter file mode of the disk that was accessed in the previous step.

B.1.1 Regenerating Compiler Modules

To fix bugs that are discovered in the compiler often requires modules of the compiler to be recompiled.[19] To replace a compiled module (a text deck) of the compiler, execute the following two commands:

```
TXTLIB DEL PASCALx AMPxcccc
TXTLIB ADD PASCALx AMPxcccc
```

where "PASCALx" is either PASCALL, PASCALO, or PASCALT, depending on which phase of the compiler is being fixed; "AMPxcccc" is the module name being replaced.

After the appropriate text modules have been replaced, then the associated load module will need to be regenerated. To regenerate PASCALL MODULE, execute the following:

```
PASCMOD AMPLMAIN PASCALL (NAME PASCALL
```

To regenerate PASCALO MODULE, execute the following:

```
PASCMOD AMPOMAIN PASCALO (NAME PASCALO
```

To regenerate PASCALT MODULE, execute the following:

```
PASCMOD AMPTMAIN PASCALT (NAME PASCALT
```

[18] 800 byte blocks are assumed. This amount is equivalent to 9 cylinders on a 3330 disk.

[19] The Pascal/VS compiler is written entirely in Pascal/VS and is self-compiling.

```
//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 97. 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 97 may serve as a model job to retrieve this file. DD operands which are **highlighted** 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 98 on page 172, Figure 99 on page 173, and Figure 100 on page 174 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
  in STEP4: 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 -- SAMPLE.PASCAL
//*
//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,(50,10,3))
//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.PASCALVS.LOAD,DISP=(NEW,CATLG),
//          DCB=(BLKSIZE=13030,RECFM=U,DSORG=PO),
//          UNIT=3330,VOL=SER=DISKVOL,
//          SPACE=(TRK,(9,1,7))
```

Figure 98. Sample installation job: (continued in Figure 99 on page 173)

```
//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,(4,1,3))
//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,(3,1,5))
//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,(2,2,2))
//FILE9   DD DSN=PASCALVS.PROCLIB,
//           VOL=REF=*.STEP1.SYSUT1,
//           UNIT=TAPE,LABEL=(9,NL),
//           DCB=BLKSIZE=3120,
//           DISP=(OLD,PASS)
//DS10    DD DSN=SYS1.SAMPLE.PASCAL,DISP=(NEW,CATLG),
//           DCB=(BLKSIZE=3120,RECFM=FB,LRECL=80,DSORG=PO),
//           UNIT=3330,VOL=SER=DISKVOL,
//           SPACE=(TRK,(5,2,2))
//FILE10  DD DSN=SAMPLE.PASCAL,
//           VOL=REF=*.STEP1.SYSUT1,
//           UNIT=TAPE,LABEL=(10,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
/*
```

Figure 99. Sample installation job: (continued in Figure 100 on page 174)

```
//*
//*    FILE 11-- PASCALVS MESSAGES
//*  (Must be stored unblocked because of BDAM access requirements)
//*
//STEP4  EXEC PGM=IEBGENER
//SYSPRINT   DD SYSOUT=*
//SYSUT1 DD DSN=PASCALVS.MESSAGES,
//          VOL=REF=*.STEP1.SYSUT1,
//          UNIT=TAPE,LABEL=(11,NL),
//          DCB=(LRECL=64,RECFM=FB,BLKSIZE=3200,DEN=3),
//          DISP=(OLD,PASS)
//SYSUT2 DD DSN=SYS1.PASCALVS.MESSAGES,DISP=(NEW,CATLG),
//          DCB=(LRECL=64,RECFM=F,BLKSIZE=64),
//          UNIT=3330,VOL=SER=DISKVOL,
//          SPACE=(TRK,(1,1))
//SYSIN  DD DUMMY
```

Figure 100. Sample installation job: (continued from Figure 98 on page 172 and Figure 99)

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 24.

B.3 LOADING THE SOURCE UNDER CMS

The compiler source is stored on the distribution tape beginning at file 13; that is, 12 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 35 free cylinders of 3330 storage.[20]
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.

[20] This is roughly 9400 800-byte blocks. Once the source files have been installed, you may find it desirable to pack them in order to save disk storage.

B.4 LOADING THE SOURCE UNDER VS2

The compiler source is stored on the distribution tape beginning at file 13. 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.
- If you do not want a listing of the source, then DDname SYSPRINT should be assigned to DUMMY in each of the job steps.
- 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.

```
//LOADSRC  JOB ,REGION=50K
//*
//*    FILE 13 -- PASCALL PASCAL - PASS 1 SOURCE (COMPILER)
//*
//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=(13,NL),
//           DISP=(OLD,PASS),
//           DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DEN=3)
//SYSPRINT DD SYSOUT=*
//*
//*    FILE 14 -- PASCALO PASCAL  - PASS 2 SOURCE (OPTIMIZER)
//*
//STEP2 EXEC PGM=IEBUPDTE,PARM=NEW
//SYSUT2  DD DSN=SOURCE.PASCALO.PASCAL,DISP=(NEW,CATLG),
//           UNIT=3330,DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB),
//           VOL=SER=DISKVOL,SPACE=(TRK,(40,10,5))
//SYSIN   DD UNIT=TAPE,VOL=REF=*.STEP1.SYSIN,LABEL=(14,NL),
//           DISP=(OLD,PASS),
//           DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DEN=3)
//SYSPRINT DD SYSOUT=*
//*
//*    FILE 15 -- PASCALT PASCAL  - PASS 3 SOURCE (TRANSLATOR)
//*
//STEP3 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=(15,NL),
//           DISP=(OLD,PASS),
//           DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DEN=3)
//SYSPRINT DD SYSOUT=*
//*
//*    FILE 16 -- PASCALD PASCAL - DEBUG SOURCE
//*
//STEP4 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=(16,NL),
//           DISP=(OLD,PASS),
//           DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DEN=3)
//SYSPRINT DD SYSOUT=*
//*
//*    FILE 17 -- PASCALX PASCAL - RUN TIME ENVIRONMENT SOURCE
//*
//STEP5 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=(17,NL),
//           DISP=(OLD,PASS),
//           DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DEN=3)
//SYSPRINT DD SYSOUT=*
```

Figure 101. Listing of the JCL to copy source files from tape: this job is stored as file 2 of the distribution tape. (continued in Figure 102 on page 177).

```
//*
//*    FILE 18 -- PASCALZ ASM - RUN TIME ENVIRONMENT SOURCE
//*
//STEP6 EXEC PGM=IEBUPDTE,PARM=NEW
//SYSUT2  DD DSN=SOURCE.PASCALZ.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=(18,NL),
//           DISP=(OLD,PASS),
//           DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DEN=3)
//SYSPRINT DD SYSOUT=*
//*
//*    FILE 19 -- MACLIBL PASCAL  - %INCLUDE LIBRARY FOR COMPILER
//*
//STEP7 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=(19,NL),
//           DISP=(OLD,PASS),
//           DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DEN=3)
//SYSPRINT DD SYSOUT=*
//*
//*    FILE 20 -- MACLIBO PASCAL  - %INCLUDE LIBRARY FOR OPTIMIZER
//*
//STEP8 EXEC PGM=IEBUPDTE,PARM=NEW
//SYSUT2  DD DSN=SOURCE.MACLIBO.PASCAL,DISP=(NEW,CATLG),
//           UNIT=3330,DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB),
//           VOL=SER=DISKVOL,SPACE=(TRK,(5,2,3))
//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
//*
//STEP9 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
//*
//STEP10 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
//*
//STEP11 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 102. Listing of the JCL to copy source files from tape: (continued from Figure 101)

APPENDIX C. ADDITIONAL LIBRARY PROCEDURES AND FUNCTIONS

In addition to the routines described in Pascal/VS Reference Manual, order number SH20-6168-1, there are several other routines which are not predefined but are provided in the Pascal/VS execution library. These routines are :

- ITOHS Procedure
- CMS Procedure
- LPAD Procedure
- RPAD Procedure
- PICTURE Function

C.1 CMS PROCEDURE

Invoke a CMS Command

Definition:

```
procedure CMS(
  const S       : STRING;
  var   RC      : INTEGER);
  EXTERNAL;
```

Where:

S is a STRING that is to be executed.
RC is the return code.

The STRING specified by S will be passed to CMS (via SVC 202) to be executed; the command must be executable in the transient area or in a shared segment. You must code the declaration as shown above, or use the INCLUDE member named "CMS" which is provided in the Pascal/VS library. This procedure is applicable under CMS only.

```
%INCLUDE CMS
  ...
CMS('CP Q T', RET);
```

C.2 ITOHS FUNCTION

Convert an INTEGER to a hex string

Definition:

```
function ITOHS(
         I      : INTEGER)
                : STRING(8);
  EXTERNAL;
```

Where:

I is the value to be converted.

This function converts the parameter I into a STRING that contains the hexadecimal representation of the integer. You must code the declaration as shown above, or use the INCLUDE member named "CONVERT" which is provided in the Pascal/VS library.

```
%INCLUDE CONVERT
  ...
WRITELN('The value ',I:0,
        ' is ',      ITOHS(I),
        ' in hexadecimal.');
```

C.3 LPAD PROCEDURE

Pads or truncates a string on the left

Definition:

```
procedure LPAD(
  var S       : STRING;
      L       : INTEGER;
      C       : CHAR);
  EXTERNAL;
```

Where:

S is the STRING to be padded;
L is the final length of S;
C is the pad character.

The procedure LPAD pads or truncates string variable S on the left. If LENGTH(S) is greater than L, then the effect is to truncate characters on the left. If LENGTH(S) is less than L, then the effect is to extend S with the character C on the left. You must code the declaration as shown above, or use the INCLUDE member named "STRING" which is provided in the Pascal/VS library.

```
%INCLUDE STRING;
  ...
S := 'ABCDEF';
LPAD(S, 10, '$');
  produces '$$$$ABCDEF' in S

S := 'ABCDEF';
LPAD(S, 5, '$');
  produces 'BCDEF' in S
```

C.4 RPAD PROCEDURE

Pads or truncates a string on the right

Definition:

```
procedure RPAD(
  var S       : STRING;
      L       : INTEGER;
      C       : CHAR);
  EXTERNAL;
```

Where:

S is the STRING to be padded;
L is the final length of S;
C is the pad character.

The procedure RPAD pads or truncates string variable S on the right. If LENGTH(S) is greater than L, then the effect is to truncate characters on the right. If LENGTH(S) is less than L, then the effect is to extend S with the character C on the right. You must code the declaration as shown above, or use the INCLUDE member named "STRING" which is provided in the Pascal/VS library.

```
%INCLUDE STRING
  ...
S := 'ABCDEF';
RPAD(S, 10, '$');
  produces 'ABCDEF$$$$' in S

S := 'ABCDEF';
RPAD(S, 5, '$');
  produces 'ABCDE' in S
```

C.5 PICTURE FUNCTION

Formats a floating point value according to a "picture" format

Definition:

```
function PICTURE(
 const P : STRING;
       R : REAL): STRING(100);
  EXTERNAL;
```

Where:

P is a picture specification;
R is the number to be formatted.

The function PICTURE returns the string representation of a real number formatted according to a "picture" specification. The characters that make up the picture specification are similar to those found in PL/I and COBOL.

A declaration for PICTURE may be obtained by including the member CONVERT from the Pascal/VS library.

A picture specification may consist of two fields: a decimal field and an exponent field. The latter is optional; the first one is always required.

The decimal field may consist of two subfields: the integer part and the fractional part. The latter is optional.

Example of picture specifications:

```
$9999.V99
9V.999ES99
$ZZZ,ZZZ,ZZ9V.99
```

A picture character may be grouped into the following categories. Picture characters may be specified in lower case.

- Digit and decimal-point specifier

  9 specifies that the associated position in the data item is to contain a decimal digit.

  V divides the decimal field into two parts: the integer part and the fractional part. This character specifies that a decimal point is assumed at this position in the associated data item. However, it does not specify that an actual decimal point is to be inserted. The integer and fractional parts of the assigned value are aligned on the V character; therefore, an assigned value may be truncated or extended with zero digits at either end. (User beware!) If no V character appears, a V is assumed at the right end of of the decimal field.

- Zero suppression characters

  Z specifies a conditional digit position in the character string value and may cause a leading zero to be replaced with a blank.

  ***** specifies a conditional digit position in the character string value and may cause a leading zero to be replaced with an asterisk ('*').

  leading zeros are those that occur in the leftmost digit positions of the integer part of floating point numbers.

- Insertion character

  Insertion characters are inserted into corresponding positions in the output string provided that zero suppression is not taking place. If zeros are being suppressed when an insertion character is encountered, a blank or an asterisk will be inserted in the corresponding place in the output string, depending on whether the zero-suppression character is a Z or an asterisk (*).

  , causes a comma to be inserted into the associated position of the output string.

  . causes a point (.) to be inserted into the associated position of the output string. The character never causes point alignment in the number. That function is served soley by the character V.

  B causes a blank to be inserted into the associated position of the output string.

- Signs and currency symbol

  The sign and currency characters ('$','+','-','$') may be used in either a static or a drifting manner. The static use specifies that a sign, a currency symbol, or a blank always appears in the associated position. The drifting use specifies that leading zeros are to be suppressed.

  A drifting character is specified by multiple use of that character in a picture field.

  + specifies a plus sign character (+) if the number is >=0, otherwise it specifies a blank.

- specifies a minus sign character (-) if the number is <0, otherwise it specifies a blank.

S specifies a plus sign character (+) if the number is >=0, otherwise it specifies a minus sign character (-).

**$** specifies a dollar sign character ($).

Exponent specifiers

The characters 'E' and 'K' delimit the exponent field of a picture specification. The exponent field must always be the last field.

E specifies that the associated position contains the letter E, which indicates the start of the exponent field.

K specifies that the exponent field appears to the right of the associated position. It does not specify a character data item.

See Figure 103 for examples.

P	R	PICTURE(P,R)
'99999'	123.0	'00123'
'ZZZZ9'	123.0	' 123'
'****9'	123.0	'**123'
'ZZZZ9'	0.0	' 0'
'ZZZZZ'	0.0	' '
'****9'	0.0	'****0'
'*****'	0.0	'*****'
'S9999'	123.0	'+0123'
'+9999'	123.0	'+0123'
'+9999'	-123.0	' 0123'
'999.99'	-123.456	'001.23'
'999V.99'	123.456	'123.46'
'ZZZ,ZZZ,ZZ9'	123456.0	' 123,456'
'***,***,**9'	123456.0	'****123,456'
'-ZZ,ZZZ,ZZ9'	-123456.0	'- 123,456'
'---,---,--9'	-123456.0	' -123,456'
'$**,***,**9V.99'	123456.78	'$***123,456.78'
'$$$,$$$,$$9V.99'	123456.78	' $123,456.78'
'S9V.9999ES99'	1.23456	'+1.2346E+00'
'S9V.9999KS99'	1.23456	'+1.2346+00'
'-999.999,V99'	1234.567	'-001.234,57'
'-9.999E9'	-1234.567	'-1.235E3'
'9B9B9B9B9B9'	123456.0	'1 2 3 4 5 6'
'9.9.9.9.9.9'	12345.0	'0.1.2.3.4.5'
'99999S'	-12345.0	'12345-'
'999+'	-123.45	'123 '
'999+'	+123.45	'123+'
'ZZZ.V99'	0.12	' 12'
'ZZZV.99'	0.12	' .12'
'-9V.999ES9'	1.23E4	' 1.230E+4'
'S9999VESZ9'	-123456.0	'-1235E+ 2'
'-V.999E-99'	123456.0	' .123E 06'

Figure 103. Examples of using the PICTURE function

Virtual Machine/Personal Computer (VM/PC) is an IBM licensed program that runs on the IBM XT/370 Personal Computer. VM/PC gives you an interactive system that has the characteristics of a VM/SP Release 2 system.

This appendix gives only the basic information needed to use the Pascal/VS programming language under VM/PC. You will also need one of the following manuals: Pascal/VS Reference Manual and the Pascal/VS Programmer's Guide; order numbers are SH20-6168 and SH20-6162 respectively.

It is assumed that the user has a general knowledge of the VM/PC operating environment, and that the VM/PC system has been installed and configured. Refer to the VM/PC User's Guide for more information regarding the VM/PC system.

D.1 INTRODUCING VM/PC FOR PASCAL/VS

This appendix describes how to use the IBM Pascal/VS programming language under VM/PC.

VM/PC is an IBM licensed program that runs on the IBM XT/370 Personal Computer, as an IBM Personal Computer Disk Operating System application. VM/PC gives you an interactive system that has the characteristics of a VM/SP Release 2 system: command entry, command formats, messages, screen formats, file naming conventions, key functions and application interfaces.

To use the Pascal/VS programming language under VM/PC, a host system must be available; this is because you must copy (download) the Pascal/VS compiler and library from the host system into your local VM/PC storage. Once you have done this, you can use the product either independently of the host system, or in connection with the host system.

VM/PC lets you set up a local 370 environment in which to do your work, known as a local session. Once you have downloaded the Pascal/VS compiler and library into your local storage, you can use that product in local sessions.

VM/PC also lets you set up a 3277 or 3101 connection with a host system on a remote computer, so that your personal computer acts as a terminal on the host system; such a connection is known as a remote session. You can use the product in remote sessions as well as in local sessions. (However, see "Licensing Considerations.")

To develop Pascal/VS programs with VM/PC, you'll use both types of sessions. You can use a remote session to create and process programs on a host system, or to copy (download) the Pascal/VS compiler and library into your local VM/PC storage. Once the Pascal/VS compiler is available in local storage, you create and compile Pascal/VS programs in local sessions.

You can also mix local and remote sessions in any combination that you find efficient. For example, you could create and edit your programs in local sessions, then copy (upload) them into the host system for compilation and execution. Or you could create and compile your programs on the host system in remote sessions, and then download the object program for execution in local sessions.

The performance of Pascal/VS on VM/PC is strongly dependent upon the nature of the specific job stream, and you may very well find that VM/PC performance with the Pascal/VS compiler is affected by the storage and paging constraints imposed by the VM/PC hardware. Therefore, as compared to a typical Pascal/VS compiler, you may experience greatly extended processing times in the VM/PC environment.

D.2 LICENSING CONSIDERATIONS

You can execute a host-resident Pascal/VS compiler from a local session. The following considerations apply:

1. When you execute the Pascal/VS compiler in a local session, the compiler must be licensed for your XT/370 machine (whether or not you have downloaded the compiler into XT/370 disk storage).

2. To execute a compiled Pascal/VS object program in a local session, that was compiled and link-edited on a host system, a license is not required.

3. When you use a remote session to execute Pascal/VS object programs that were compiled on the host system, a license is not required.

D.3 USING VM/PC

Under VM/PC, you use VM/SP-CMS commands to create, modify, compile, link-edit or load, execute, debug, and test your Pascal/VS programs.

The commands most useful to you in performing these tasks are briefly described in Figure 104.

You will also find the following CP commands useful:

- LINK : which makes a device associated with another virtual machine available to your virtual machine configuration, based upon information in the user's VM/SP directory entry.
- SPOOL : which modifies the spooling control options in effect for one or more virtual spooling devices.

Command	How Used
ACCESS	Activates a virtual disk for use
EXEC	Executes a file that consists of one or more CMS commands
FILEDEF	Defines a file and its input/output devices
GLOBAL	Specifies text libraries to be searched to resolve external references in a program being loaded
INCLUDE	Specifies additional text files for use during program execution
LISTFILE	Displays a list of your files
LOAD	Places a text file in storage and establishes the linkages for execution
PRINT	Prints a file on the off-line printer
RENAME	Changes the filename, filetype, and/or filemode of a file
SET	Establishes, turns off, or resets a particular function of the CMS virtual machine
START	Begins execution of a previously loaded and link-edited program file
TYPE	Displays all or part of a file at the terminal
XEDIT	Puts you in edit mode to create and edit source program and data files and lets you use the XEDIT subcommands

Figure 104. CMS Command Summary

D.4 METHODS OF USING PASCAL/VS UNDER VM/PC

There are two different ways in which you use Pascal/VS under VM/PC:

1. Copy (download) the Pascal/VS compiler modules onto local disk files, and then invoke Pascal/VS in local sessions. (You need to download only when you first access Pascal/VS, when a new maintenance update is applied, or when a new release has been installed on the host system.)

2. Link to the host system minidisk containing Pascal/VS compiler and library, and then access it from the local session as a remote minidisk. (You must do this after every Initial Program Load (IPL) of CMS, or whenever the link to the host system is severed.)

   Depending on your link with the system, and on the system load, this often is not an efficient way to operate.

   Note As noted under "Licensing Considerations" above, your VM/PC must be licensed for Pascal/VS if you are to execute the compiler in a local session. This is true even if you do not download the compiler onto your local VM/PC storage.

D.5 DOWNLOADING THE PASCAL/VS COMPILER INTO VM/PC

To use Pascal/VS under VM/PC, you can copy (download) the Pascal/VS modules into your local files. The modules you must copy are listed in Figure 105 on page 188

Downloading is necessary only when you first access Pascal/VS, or after a new release or maintenance updates have been installed on the host system.

Both the virtual storage and minidisk storage must be allocated with approximately 1.0M bytes. These storage requirements are for Pascal/VS compiler and library only; additional storage is needed for the source and/or object program files.

Figure 106 on page 188 shows you the commands you must issue. The procedure is as follows:

1. Link (if necessary) and access the local minidisk that is the target minidisk for the copy operation. If the target minidisk is your own minidisk, the link is not required.

2. Link and access the host minidisk that contains the Pascal/VS modules.

3. Copy the Pascal/VS modules from the host minidisk to the local minidisk. (This is known as downloading.)

4. Release the host Pascal/VS minidisk; it is no longer required.

```
PASCALS    MODULE
PASCALL    MODULE
PASCALO    MODULE
PASCALT    MODULE
PASCALVS   TXTLIB
PASDEBUG   TXTLIB
PASCALVS   MACLIB
PASCALVS   EXEC
PASCMOD    EXEC
PASCALVS   MESSAGES
PASCALVS   CMSHELP
```

Figure 105. Pascal/VS Modules Needed for Downloading

```
*************************************************************************
*
* 1) Link and access the target VM/PC minidisk.
*
CP LINK vm/pc-id ttt aaa W write-password
ACCESS aaa filemode1
*
* 2) Link and access the host minidisk that contains the Pascal/VS
*    compiler and library.
*
CP LINK host-id hhh bbb RR read-password REMOTE
ACCESS bbb filemode2
*
* 3) Copy the files you need.
*
COPYFILE filename filetype filemode2 = = filemode1
     .
     .
*
* 4) Release the Pascal/VS host minidisk.
*
RELEASE filemode2 ( DET
*
*
* Where:
*   ttt - is the virtual address of the local target minidisk that
*         will store the Pascal/VS modules.
*   aaa - is an unused virtual address on the local VM/PC machine.
*   hhh - is the virtual address of the host minidisk that contains
*         the Pascal/VS modules.
*   bbb - is the virtual disk address you use to refer to the host
*         disk.
*   filemode1 - is the filemode of the target minidisk on the local
*               VM/PC machine.
*   filemode2 - is the filemode of the host minidisk that contains
*               the Pascal/VS modules.
*
*************************************************************************
```

Figure 106. CMS Commands to Download Pascal/VS From a Local Session

D.6 ACCESSING THE PASCAL/VS COMPILER ON THE HOST

The other way to use Pascal/VS under VM/PC, is to link to the host system minidisk containing Pascal/VS compiler and library and then access it from the local session as a remote minidisk.

Linking and accessing are necessary whenever there is a an Initial Program Load (IPL) of CMS, and whenever the link to the host system severed.

Depending on your link with the host system and on the system load, this often is not an efficient way to operate compared to downloading.

The virtual storage requirement is approximately 1.0M bytes, but there is no additional VM/PC minidisk storage requirement for the Pascal/VS compiler and library since it resides in the host system minidisk storage area. Additional storage is needed though for the source and/or object program files.

Figure 107 shows you the commands you must issue to link and access the host minidisk that contains the Pascal/VS modules.

```
*************************************************************************
*
* Link and access the host minidisk that contains the Pascal/VS
* compiler and library.
*
CP LINK host-id hhh bbb RR read-password REMOTE
ACCESS bbb filemodel
*
*
* Where:
*   hhh - is the virtual address of the host minidisk that contains
*         the Pascal/VS modules.
*   bbb - is the virtual disk address you use to refer to the host
*         disk.
*   filemodel - is the filemode of the local VM machine
*
*************************************************************************
```

Figure 107. CMS Commands to Access Pascal/VS From a Local Session as a Remote Minidisk

D.7 INVOKING PASCAL/VS UNDER VM/PC

You must first make Pascal/VS available on a minidisk you can access. For example:

CP LINK userid aaa aaa RR read-password
ACCESS aaa filemodel

If Pascal/VS is stored on your A-disk, or another disk you can access, you can omit the LINK and ACCESS commands. (If you must issue these commands each time you log on to VM/PC, you can put them into your PROFILE EXEC, which issues them for you.)

Next, you can invoke Pascal/VS through the following command:

PASCALVS fn [ft [fm]] [[options...[]]]

where "fn" is the name of the Pascal/VS program, "ft" is PASCAL if omitted,and "options" let you modify the default compiler options in force for your organization.

To build a load module, issue the following command:

PASCMOD main [fns...] [[options...[]]]

where "main" is the name of the main program module, "fns" are the names of segment modules and text libraries (TXTLIB's) which are to be included, and "options" allow you to override default options.

To invoke the load module, issue the following command:

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

where "modname" is the name of the load module, "rtparms" are the run time options, and "parms" are the parameters (if any) being passed to the Pascal program.

D.8 VM/PC PROCESSING RESTRICTIONS ON PASCAL/VS

The following processing capabilities are not available when you are executing an object program in a local VM/PC session:

1. Any Pascal/VS restrictions on CMS processing apply for VM/PC as well.

2. Magnetic tape file processing is not available: this means that you can not define (FILEDEF) a Pascal/VS sequential file to a magnetic tape medium.

D.9 PASCAL/VS PROGRAMMING TIPS

You can improve processing time if you specify the NOPRINT Pascal/VS compiler option that suppresses the generation of a program listing (if a listing is not required). NOPRINT automatically forces the following three compiler options to become active:

- NOSOURCE
- NOXREF
- NOLIST

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Pascal/VS Programmer's Guide Printed in U.S.A. SH20-6162-2

SH20-6162-2

READER'S COMMENT FORM

Pascal/VS 5796-PNQ
Programmer's Guide
SH20-6162-2

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name, company, mailing address, and date:

_____

_____

_____

_____

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

What is your occupation? _____

Number of latest Newsletter associated with this publication: _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

SH20-6162-2

Reader's Comment Form

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department 68Y
P.O. Box 152750
Irving, Texas 75015-2750

Fold and tape | Please Do Not Staple | Fold and tape

IBM®

SH20-6162-02

Cut or Fold Along Line