IBM

VS Pascal

SC26-4319-1

Application Programming Guide

Release 2

VS Pascal

SC26-4319-1

Application Programming Guide

Release 2

Second Edition (December 1988)

This edition replaces and makes obsolete the previous edition, SC26-4319-0.

This edition applies to Release 2 of VS Pascal, Program Number 5668-767 (Compiler and Library) and 5668-717 (Library only) and to any subsequent releases until otherwise indicated in new editions or technical newsletters.

The changes for this edition are summarized in Appendix A, "Summary of Changes" on page 259. Specific changes are indicated by a vertical bar to the left of the change. These bars will be deleted at any republication of the page affected. Editorial changes that have no technical significance are not noted.

Changes are made periodically to this publication; before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370, 30xx, 4300, and 9370 Processors Bibliography*, GC20-0001, for the editions that are applicable and current.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM"s program may be used. Any functionally equivalent program may be used instead.

Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality. If you request publications from the address given below, your order will be delayed because publications are not stocked there.

A Reader's Comment Form is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Programming Publishing, P. O. Box 49023, San Jose, California, U.S.A. 95161-9023. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Preface

VS Pascal Application Programming Guide describes how to use the VS Pascal compiler and explains how to compile, link-edit, run, and debug VS Pascal programs. It provides information about how to use registers, storage, input/output facilities, and how to communicate with other programming languages.

This book contains no definition of the VS Pascal programming language and its syntax. For information about this subject matter, see *VS Pascal Language Reference*.

How to Use this Manual

VS Pascal Application Programming Guide is divided into two major parts: Guide and Reference. Of the first three chapters in the Guide section, you need read only the chapter that pertains to your operating environment. The rest of the chapters in the Guide, along with the Reference chapters, are relatively self-contained, and you can consult them as the need arises. Before you consult the Reference chapters, you might want to read about syntax diagrams in Appendix B, "How to Read Syntax Diagrams" on page 263.

Part I. Guide

Chapter 1, "How to Run a Program under VM" on page 3, explains how to compile, link-edit, and run a program under VM.

Chapter 2, "How to Run a Program under MVS/TSO" on page 13, explains how to compile, link-edit, and run a program under MVS/TSO.

Chapter 3, "How to Run a Program in an MVS Batch Environment" on page 23, explains how to compile, link-edit, and run a program in MVS batch mode using cataloged procedures.

Chapter 4, "How to Read VS Pascal Listings" on page 33, describes how to read VS Pascal source, cross-reference, and other listings.

Chapter 5, "How to Use the Input/Output Facilities" on page 45, describes how VS Pascal passes data to files.

Chapter 6, "VS Pascal Run-Time Error Handling" on page 73, describes run-time error handling in VS Pascal and explains how you can write a routine to control run-time error handling.

Chapter 7, "How to Debug Your Program" on page 83, describes how to use the VS Pascal Interactive Debugging Tool, using debugging output from a sample program.

Chapter 8, "How to Use Interlanguage Communication" on page 97, describes how VS Pascal can call and be called by programming languages such as PL/I, COBOL, and FORTRAN.

Chapter 9, "Interfacing With IMS" on page 121, explains how to use the GENERIC directive to interface with IMS.

Appendix F, "Double-Byte Character Set (DBCS) Support" on page 281, describes how VS Pascal supports double-byte character set (DBCS) data.

Appendix G, "Migration Considerations" on page 283, details the migration considerations (1) between VS Pascal Release 1 and Release 2 and (2) between Pascal/VS and VS Pascal Release 1.

Consult the "Bibliography" on page 295 for other useful publications.

Industry Standards

The VS Pascal Compiler and Library, Release 2, supports the specifications of the American National Standard Pascal Computer Programming Language (ANSI/IEEE770X3.97-1983). This standard is referred to in this manual as the ANSI Standard, or simply Standard Pascal. For more information on VS Pascal and Standard Pascal, see Appendix D, "VS Pascal and the 1983 ANSI/IEEE Pascal Standard" on page 271.

VS Pascal supports the specifications of the ISO Programming Language standard (ISO 7185-1983) at level 0. This standard is referred to as the ISO standard in this manual. VS Pascal also supports the Federal Information Processing Standard Publication (FIPS PUB) 109.

Contents

Figures

Part I. Guide

Chapter 1. How to Run a Program under VM

This chapter describes how to run a VS Pascal program under VM. If you are not using VS Pascal under VM, you can skip this chapter.

There are four steps to running a VS Pascal program under VM:

1. Compile the program to produce an object module.
2. Generate a load module from the object module.
3. Define all files used within the program.
4. Invoke the load module.

VS Pascal requires a virtual machine of at least 1M to compile a program. Execution of a compiled program can be performed in an 864K CMS machine.

Step 1: How to Compile a Program

The standard method of invoking the VS Pascal compiler under VM is by using the VSPASCAL EXEC. See "VSPASCAL EXEC" on page 127 for the full description of the VSPASCAL EXEC.

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

```
VSPASCAL fn ft fm
```

Where	Represents
fn	The file name of the program.
ft	The file type. This is optional; when you do not specify a file type, VS Pascal assumes the file type is PASCAL.
fm	The file mode. This is also optional; VS Pascal assumes a default file mode of "*".

The compiler translates a source program into object code, which is stored 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 residing in a file called SORT PASCAL, issue the command:

```
VSPASCAL SORT
```

If the program compiles without error, the file named SORT TEXT contains the object code.

For Programs that Use the %INCLUDE Compiler Directive

The %INCLUDE directive in a source program identifies a member from a macro library (MACLIB) and directs the compiler to search for that member. When found, that MACLIB member becomes the input stream for the compiler. After the compiler has read the entire member, it returns to the previous input stream, at the point immediately after the %INCLUDE directive.

At compile time, you must use the LIB parameter of the VSPASCAL EXEC to tell the compiler which MACLIBs it must search to find members named by a %INCLUDE. (See "VSPASCAL EXEC" on page 127.)

The order in which MACLIBs are identified using the LIB parameter of the VSPASCAL EXEC can affect program results. The %INCLUDE directive can identify a MACLIB member in one of two ways:

- *By naming both the MACLIB ddname, and the member.* In this case, before compiling the program, you must use the FILEDEF command to associate the ddname of the MACLIB with the MACLIB. This way, the compiler searches only the MACLIB named.
- *By naming only the member.* In this case, the compiler searches the default library, SYSLIB, which contains all the MACLIBs *in the order that you listed them* on the LIB parameter of the VSPASCAL EXEC. When a %INCLUDE identifies a member name that exists in two or more MACLIBs, the compiler uses the first so-named member that it encounters.

As an example, suppose the LIB option of the VSPASCAL EXEC lists, in order, LIB1, LIB2, and LIB3. Both LIB2 and LIB3 contain a member named CONSTANTS. When the compiler encounters a %INCLUDE CONSTANTS (without an accompanying MACLIB name), it first searches LIB1. Not finding CONSTANTS in LIB1, it goes to LIB2, where it finds and uses CONSTANTS. If the program must use CONSTANTS from LIB3 instead, you must either (1) change the %INCLUDE directive in the source to specify both the ddname of the MACLIB *and* the member name or (2) list LIB3 before LIB2 in the LIB option of the VSPASCAL EXEC at compile time.

The IBM-supplied default library is VSPASCAL, which you can think of as residing at the "bottom" of SYSLIB. Therefore, VSPASCAL is always the last MACLIB searched for any member named on a %INCLUDE, and you do not need to specify it on the VSPASCAL EXEC or define it with FILEDEF.

Note: When a %INCLUDE specifically names VSPASCAL, you must define it with the FILEDEF command before compile time.

Passing Compile-Time Options

Compile-time options (see Chapter 13, "Compile-Time Options" on page 155) are parameters passed to the compiler specifying whether or not a particular feature is to be active. You can specify a list of compile-time options with the VSPASCAL EXEC (see "VSPASCAL EXEC" on page 127). You must precede the options list with a left parenthesis.

For example, to compile the program TEST PASCAL with the Interactive Debugging Tool enabled (DEBUG) and without a cross-reference table (NOXREF), issue the following command:

```
VSPASCAL TEST ( DEBUG NOXREF
```

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. The listing is placed in a file *fn* LISTING, where *fn* is the file name of the unit being compiled.

For a detailed description of the information in the source listing, see "Source Listing" on page 34.

Compiler Diagnostics

Any compiler-detected errors in your program are displayed on your terminal, or written to a disk file named *fn* CONSOLE (where *fn* is the file name of the unit being compiled) if the CONSOLE option is specified. The errors are also 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 being scanned by the compiler is displayed on your console. Below the printed line, a plus sign (+) 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 listing, a diagnostic message is produced for each error number.

For a list of compiler-generated messages, see "Compiler Messages—Source Code Processing" on page 216.

Figure 1 shows an example of a VS Pascal program and the corresponding compiler diagnostic. The WRITELN statement contains an error, so that you can see how the compiler diagnostic flags and records an error.

VS Pascal Program Containing an Error

```
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.
```

VS Pascal Compiler Diagnostic

```
INVOKING VS PASCAL RELEASE 2.0
 STARTING LANGUAGE ANALYSIS PASS...
     WRITELN(OUTFILE BUFFER)
                           +41
ERROR  41: COMMA "," EXPECTED
NUMBER OF ERRORS REPORTED:   1

SOURCE LINES:  14; COMPILE TIME:  0.13 SECONDS; COMPILE RATE:  6462 LPM

R(00008);
```

Figure 1. VS Pascal Program Containing an Error and Corresponding Compiler Diagnostic

Cross-System Compilation

You can compile a VS Pascal program under VM and link-edit and run the object deck under MVS provided you do not use system-specific features such as the CMS procedure or the NAME file opening option.

Step 2: How to Build a Load Module (Link-Editing)

The PASCMOD EXEC generates load modules from VS Pascal object code. It link-edits *text decks* (a main program unit and any segment units of file type TEXT) together with *text libraries* (the VS Pascal run-time library and any other libraries of compiled code with the file type TXTLIB). The VS Pascal run-time environment resides in the VSPASCAL and AMPLANG TXTLIBs, which are included by the PASCMOD EXEC. Link-editing produces a load module whose file name matches the file name of the main program unit and whose file type is MODULE.

If your program consists of just one source unit (that is, you have no segment units), you generate a load module by invoking PASCMOD with the name of the program. For example, to generate a load module for a compiled program named SORT TEXT, enter:

```
PASCMOD SORT
```

The resulting load module is SORT MODULE.

If your program consists of two or more source units, list the name of the main program unit, followed by the names of all needed segment units and text libraries. For example, to build a load module for a compiled program that resides in three text decks (program unit MAIN TEXT, segment unit ASEG TEXT, and segment unit BSEG TEXT) and calls routines in a text library called UTILITY TXTLIB, use:

```
PASCMOD MAIN ASEG BSEG UTILITY
```

The resulting load module is MAIN MODULE.

For a description of the syntax and options available with PASCMOD, see "PASCMOD EXEC" on page 128.

For programs to be executed with the DEBUG option, the VS Pascal run-time debugging library PASDEBUG TXTLIB must be present. (For information on the Interactive Debugging Tool, see Chapter 7, "How to Debug Your Program" on page 83.)

For programs to be executed under transient run-time: Any program that you intend to run with the transient library option must be link-edited with the TRANLIB option specified on the PASCMOD EXEC.

For example, to compile the program TEST PASCAL for execution with the transient run-time library, use:

```
PASCMOD TEST (TRANLIB
```

The default is NOTRANLIB, which statically link-edits the load module for standard execution.

Combining Release 1 and Release 2 Code

When combining units compiled with VS Pascal Release 1 and VS Pascal Release 2, you must link-edit the modules with the VS Pascal Release 2 libraries.

Because some debugger tables have been changed to allow larger programs to be debugged, units compiled with DEBUG under Release 1 must be recompiled using Release 2.

Step 3: How to Define Files

Before you invoke the generated load module, you must define the files that your program requires. You do this with the CMS command FILEDEF. See the appropriate VM publication for a description of the FILEDEF command ("VM" on page 295).

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 VS Pascal declaration in Figure 2 are SYSIN, SYSPRINT, and OUTPUTFI, respectively. Figure 2 also shows the FILEDEF commands required for each of the three file variables, along with INPUT and OUTPUT.

File Declarations As They Appear in a VS Pascal Program:

```
VAR
   SYSIN,
   SYSPRINT   : TEXT;
   OUTPUTFILE : FILE OF INTEGER;
```

FILEDEF Commands Required for Those Declarations:

```
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 2. Sample File Declarations with Complementary CMS FILEDEF Commands

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

For files that need to be opened for output, the LRECL, BLKSIZE, or RECFM are assigned default values if not specified. For a description of the defaults, see "Data Set DCB Attributes" on page 46.

Step 4: How to Invoke the Load Module

After creating the load module and defining the files it needs, you are ready to run the program. You do this by invoking the module.

Invoking the Module Using PASCRUN

The PASCRUN EXEC invokes a precompiled module and issues a GLOBAL TXTLIB AMPLANG and a GLOBAL LOADLIB AMPXLVEC PASRTLIB. AMPLANG TXTLIB is needed for the LANGUAGE run-time option, PASRTLIB LOADLIB is needed for the XA and TRANLIB link-edit options, and AMPXLVEC LOADLIB is needed for the TRANLIB link-edit option.

```
PASCRUN modname options / parameters
```

Where	Represents
PASCRUN	The name of the command.
modname	The name of the load module.
options	The run-time options.
parameters	The parameters (if any) being passed.

Run-time options are also passed as a parameter list. To distinguish run-time options being passed to the VS Pascal environment from parameters that your program reads (via the PARMS function), you must end the run-time option list with a slash. The program parameters, if any, must follow the slash as shown below.

```
PASCRUN modname options / parameters
```

For a description of run-time options, see Chapter 14, "Run-Time Options" on page 167.

For a description of PASCRUN, see "PASCRUN EXEC" on page 130.

Transient Run-Time Considerations

Before invoking a program that was link-edited with the TRANLIB option, you must make sure that the run-time libraries referenced in the program have been identified.

Ensure the GLOBAL LOADLIB CMS command is issued to identify the AMPXLVEC and PASRTLIB LOADLIBs.

Sample VM Session: Compiling, Link-Editing, and Running a Program

Figure 3 shows how to compile, link-edit, and execute an existing source module. The commands you enter from the terminal are preceded with =>.

```
    READY;

 => VSPASCAL LANDER (PRINT LIST

    INVOKING VS PASCAL RELEASE 2.0
     STARTING LANGUAGE ANALYSIS PASS...
     STARTING OPTIMIZATION PASS...
     STARTING CODE GENERATION PASS...
     NO COMPILER DETECTED ERRORS
    SOURCE LINES:  47; COMPILE TIME:  0.19 SECONDS; COMPILE RATE: 15032

    READY;

 => PASCMOD LANDER
    READY;

 => FILEDEF INPUT TERM
    READY;

 => FILEDEF OUTPUT TERM
    READY;

 => LANDER PARMS GO HERE
```

Figure 3. Compiling, Link-Editing, and Executing a Program under VM

31-Bit Addressing Mode

Programs compiled by VS Pascal can execute in the 31-bit addressing mode of the VM/XA operating system. With 31-bit addressing, you have more freedom to define or reference larger data areas, files, and tables, and to create a larger overall program. Your program and its data are no longer constrained to fit in a 16-megabyte address space. However, no single record or array can be greater than 16 megabytes.

Modules that are link-edited together must also reside in the same address space (either above or below the 16-megabyte address line).

To take advantage of 31-bit addressing, you must either:

- Generate your load module using the XA option in the PASCMOD EXEC
- Use the AMODE and RMODE options when loading your program and generating your load module.

Every program that executes in VM/XA is assigned two attributes: AMODE (addressing mode) and RMODE (residence mode). AMODE is the attribute of an entry point into a load module that specifies the addressing mode in effect when the load module is entered at run-time. RMODE is the attribute of a load module that specifies the residence mode of a load module when it is loaded into virtual storage for execution.

Valid AMODE and RMODE specifications are shown in Figure 4.

Attribute	Meaning
AMODE=24	24-bit addressing mode
AMODE=31	31-bit addressing mode
AMODE=ANY	Either 24-bit or 31-bit addressing mode
RMODE=24	The module must reside in virtual storage below 16 megabytes. Use RMODE=24 for 31-bit programs that have 24-bit dependencies.
RMODE=ANY	Indicates that the module can reside anywhere in storage.

Figure 4. AMODE and RMODE Specifications

For more information on AMODE and RMODE, see *VM/XA System Product CMS Application Program Development Guide*, SC23-0355.

Sample VM Session to Invoke 31-Bit Addressing Mode

Figure 5 shows how to compile, link-edit, and execute an already existing source module in 31-bit addressing mode under VM. The commands entered from the terminal are preceded with =>.

```
    READY;

 => VSPASCAL LANDER (PRINT LIST

    INVOKING VS PASCAL RELEASE 2.0
     STARTING LANGUAGE ANALYSIS PASS...
     STARTING OPTIMIZATION PASS...
     STARTING CODE GENERATION PASS...
     NO COMPILER DETECTED ERRORS
    SOURCE LINES:  47; COMPILE TIME:  0.19 SECONDS; COMPILE RATE: 15032

    READY;

 => PASCMOD LANDER (XA
    READY;

 => FILEDEF INPUT TERM
    READY;

 => FILEDEF OUTPUT TERM
    READY;

 => LANDER PARMS GO HERE
```

Figure 5. Compiling, Link-Editing, and Running a Program under VM/XA

Chapter 2. How to Run a Program under MVS/TSO

This chapter describes how to compile and execute a VS Pascal program under the Time Sharing Option (TSO) of MVS. If you are not using VS Pascal under TSO, then you can skip this chapter.

There are four steps to running a VS Pascal program:

1. Compile the program to produce an object module.
2. Generate a load module from the object module.
3. Allocate all data sets used within the program.
4. Invoke the load module.

VS Pascal requires a minimum region size of 512K to compile a program under MVS. A compiled and link-edited program can execute in a 448K region.

Step 1: How to Compile a Program

The VS Pascal compiler is invoked under TSO by means of a CLIST. IBM provides a sample CLIST named VSPASCAL to compile a VS Pascal program. For a detailed description of the VSPASCAL CLIST parameters, see "VSPASCAL CLIST" on page 131.

To invoke the VS Pascal compiler with the conditions:

- The user identification is ABC.
- The data set containing the program is named ABC.SORT.PASCAL.
- The compiler listing is to be directed to the printer.
- The default options and data set names are to be used.

Issue the following command:

```
VSPASCAL SORT SYSPRINT(A)
```

To invoke the VS Pascal compiler with the conditions:

- The user identification is XYZ.
- The 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.
- The default options and data set names are to be used for the rest.

issue the following command:

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

For Programs that Use the %INCLUDE Compiler Directive

The %INCLUDE directive in a source program identifies a member from a partitioned data set and directs the compiler to search for that member. When found, that data set member becomes the input stream for the compiler. After the compiler has read the entire member it returns to the previous input stream at the point immediately after the %INCLUDE directive.

At compile-time, you must use the LIB parameter of the VSPASCAL CLIST to tell the compiler in which data sets it must search to find members named by a %INCLUDE. (See "VSPASCAL CLIST" on page 131.)

The order in which data sets are identified, using the LIB parameter of the VSPASCAL CLIST, can affect program results. To understand why, remember that the %INCLUDE directive can identify a data set member in one of two ways:

- *By naming both the ddname of the data set and the member.* In this case, the compiler searches only the data set named. Before compiling the program, use the TSO ALLOCATE command to associate the ddname of the data set with the data set.
- *By naming only the member.* In this case, the compiler searches the default library, SYSLIB, which contains all the data sets *in the order that you listed them* on the LIB parameter of the VSPASCAL CLIST. When a %INCLUDE identifies a member name that exists in two or more data sets, the compiler uses the first so-named member that it encounters.

As an example, suppose the LIB option of the VSPASCAL CLIST lists, in order, LIB1, LIB2, and LIB3. Both LIB2 and LIB3 contain a member named CONSTANTS. When the compiler encounters a %INCLUDE CONSTANTS (without an accompanying data set name), it first searches LIB1. Not finding CONSTANTS in LIB1, it goes to LIB2, where it finds and uses CONSTANTS. If the program must use CONSTANTS from LIB3 instead, you must either (1) change the %INCLUDE directive in the source to specify both the ddname of the data set *and* member name or (2) list LIB3 before LIB2 on the LIB option of the VSPASCAL CLIST at compile-time.

The IBM-supplied default library for MVS/TSO is VSPASCAL.VSPV1R2.SAMPMAC1, which you can think of as residing at the "bottom" of the of SYSLIB. Therefore, VSPASCAL is always the last data set searched for any member named on a %INCLUDE, and you do not need to specify it on the VSPASCAL CLIST, or in the LIB list.

Note: When a %INCLUDE specifically names VSPASCAL, you must define VSPASCAL with the ALLOCATE command before compile time.

To invoke the VS Pascal compiler with the conditions:

- The user identification is P123.
- The 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 and VSPASCAL.VSPV1R2.SAMPMAC1.
- The default options and data set names are to be used for the rest.

Issue the following command:

```
VSPASCAL MAIN LIB('PASLIB,''VSPASCAL.VSPV1R2.SAMPMAC1''')
```

Note: The high-level qualifier names (VSPASCAL.VSPV1R2) might be different at your site.

Passing Compile-Time Options

Compile-time options (see Chapter 13, "Compile-Time Options" on page 155) are parameters passed to the compiler specifying whether or not a particular feature is to be active. You can specify a list of compile-time options on the VSPASCAL CLIST (see "VSPASCAL CLIST" on page 131).

For example, to compile the program TEST PASCAL with the Interactive Debugging Tool enabled (DEBUG) and without a cross-reference table (NOXREF), issue the command:

```
VSPASCAL TEST DEBUG NOXREF
```

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.

If you use the VSPASCAL CLIST ("VSPASCAL CLIST" on page 131), remember that the default is to suppress the compiler listing. If you want a compiler listing, you can specify

- PRINT(*), which will display the listing on your terminal.
- PRINT(*dsname*), which will write the listing on the data set named *dsname*.
- SYSPRINT(*sysclass*), which will write the listing to the SYSOUT class named *sysclass*.

For a detailed description of the information in the source listing, see Chapter 4, "How to Read VS Pascal Listings" on page 33.

Compiler Diagnostics

By default, compiler diagnostics are displayed on your terminal. If the CONSOLE(*dsname*) operand appears on the VSPASCAL CLIST, the diagnostics are stored in a data set instead. The errors are also 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 being scanned by the compiler is printed on your terminal (or written to the CONSOLE data set). Underneath the printed line, a plus sign (+) 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 listing, a diagnostic message is produced for each error number.

Compiler-generated messages are listed and explained in "Compiler Messages—Source Code Processing" on page 216.

Figure 6 shows a VS Pascal program and a corresponding compiler diagnostic. The WRITELN statement contains an error so that you can see how the compiler diagnostic flags and reports errors.

VS Pascal Program Containing an Error

```
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.
```

VS Pascal Compiler Diagnostic

```
INVOKING VS PASCAL RELEASE 2.0
 STARTING LANGUAGE ANALYSIS PASS...
     WRITELN(OUTFILE BUFFER)
                            +41
 ERROR  41: COMMA "," EXPECTED
 NUMBER OF ERRORS REPORTED:   1

SOURCE LINES:  14; COMPILE TIME:  0.13 SECONDS; COMPILE RATE: 6462 LPM
RETURN CODE: 8
```

Figure 6. VS Pascal Program Containing an Error and Corresponding Compiler Diagnostic

Cross-System Compilation

You can compile a VS Pascal program under MVS and link-edit and run the object module under VM.

Step 2: How to Build a Load Module (Link-Editing)

To generate a load module from a VS Pascal object module, you can use either the TSO LINK command or the PASCMOD CLIST (see "PASCMOD CLIST" on page 134). The CLIST performs the same function as the LINK command, and it automatically includes the VS Pascal run-time library in generating the load module. Also, if you are using the Interactive Debugging Tool, the CLIST includes the VS Pascal debugging library.

Every VS Pascal object module contains references to the run-time routines. These routines are stored in two libraries called:

```
VSPASCAL.VSPV1R2.SAMPRUN1
VSPASCAL.VSPV1R2.SAMPMSG1
```

Note: The high-level qualifier names (VSPASCAL.VSPV1R2) might be different at your installation.

These libraries must be linked into a VS Pascal object module in order to resolve all external references properly. If the PASCMOD CLIST is used, these libraries are included automatically.

Example of Building a Load Module: In this example, a load module is created from a compiled VS Pascal program consisting of three object modules. The conditions are:

- The user-identification is ABC.
- The data sets containing the three object modules are:

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

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

You enter:

```
PASCMOD * LOAD(PROGS(SORT))  OBJECT('SORT,SEG1,SEG2')
```

The system prompts you with:

```
ENTER CONTROL CARDS
```

You respond with:

```
ENTRY VSPASCAL
```

The system responds with the READY message.

For Programs to be Executed with the DEBUG Option: If you are using the Interactive Debugging Tool, then you must include the debugging library containing the debugging modules. The name of this library is:

```
VSPASCAL.VSPV1R2.SAMPDBG1
```

Note: The high-level qualifier names (VSPASCAL.VSPV1R2) might be different at your installation.

This library must be specified before the run-time library. If the PASCMOD CLIST is used, this library is included if the option DEBUG is specified.

If more than one object module is linked together, you must specify an entry point by means of a linkage editor control statement. The name of the entry point for any VS Pascal program is VSPASCAL.

For Programs to be Executed Under Transient Run-Time: Any program that you intend to run with the transient run-time option must be link-edited using the TRANLIB option.

For example, to link-edit the program TEST PASCAL for transient run-time execution, use:

```
PASCMOD * OBJECT('TEST') TRANLIB
```

The default is NOTRANLIB, which statically link-edits the load module for standard execution.

Combining Release 1 and Release 2 Code

When combining units compiled with VS Pascal Release 1 and VS Pascal Release 2, you must link-edit the modules with the VS Pascal Release 2 libraries.

Because some debugger tables have been changed to allow larger programs to be debugged, units compiled with DEBUG under Release 1 must be recompiled using Release 2.

Step 3: How to Define Files

Before you invoke the generated load module, you must define the files that your program requires. You do this with the TSO command ALLOCATE (or ALLOC). For more information on the ALLOCATE command, see the appropriate TSO manual (see "TSO" on page 295).

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 VS Pascal declaration in Figure 7 are SYSIN, SYSPRINT, and OUTPUTFI, respectively. Figure 7 also shows the ALLOC commands required for each of the three file variables, along with INPUT and OUTPUT.

File Declarations As They Appear in a VS Pascal Program:

```
VAR
  SYSIN,
  SYSPRINT   : TEXT;
  OUTPUTFILE : FILE OF INTEGER;
```

ALLOC Commands Required for Those Declarations:

```
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. Sample File Declarations and Complementary TSO ALLOC Commands

For files that need to be opened for output, the LRECL, BLKSIZE, or RECFM are assigned default values if not specified via the ATTR command. For a description of the defaults, see "Data Set DCB Attributes" on page 46.

Step 4: How to Invoke the Load Module

After creating the module and defining the files, you are ready to execute the program. You do this through the TSO CALL command (see "CALL Command" on page 136).

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

```
CALL modname 'parameters'
```

Where	Represents
modname	The name of the load module.
parameters	The parameters (if any) being passed.

If you are using MVS/XA, you must include PASRTLIB in the STEPLIB for your TSO session before issuing the CALL command.

Run-time options are also passed as a parameter list. To distinguish run-time options being passed to the VS Pascal environment from parameters that your program reads (via the PARMS function), you must end the run-time parameter list with a slash. The program parameters, if any, must follow the slash as shown below.

```
CALL  modname 'runtimeparameters / parameters'
```

If you are specifying the LANGUAGE run-time option, you must include VSPASCAL.VSPV1R2.SAMPMSG1 in your STEPLIB prior to running the program.

For a description of run-time options, see Chapter 14, "Run-Time Options" on page 167.

Transient Run-Time Considerations

Before invoking a program that was link-edited with the TRANLIB option, you must make sure that the run-time libraries referenced in the program have been identified.

Your logon procedures must reference the run-time load libraries in the STEPLIB DD card.

Sample TSO Session: Compiling, Link-Editing, and Running a Program

Figure 8 shows how to compile, link-edit, and execute an existing source module. The commands you enter from the terminal are preceded with =>.

```
    READY

 => VSPASCAL LANDER SYSPRINT(A) LIST

    INVOKING VS PASCAL RELEASE 2.0
     STARTING LANGUAGE ANALYSIS PASS...
     STARTING OPTIMIZATION PASS...
     STARTING CODE GENERATION PASS...
     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 8. Compiling, Link-Editing, and Executing a Program under MVS

31-Bit Addressing Mode

Programs compiled by VS Pascal can execute in the 31-bit addressing mode of the MVS/XA and MVS/ESA operating systems. With 31-bit addressing, you have more freedom to define or reference larger data areas, files, and tables, and to create a larger overall program. Your program and its data are no longer constrained to fit in a 16-megabyte address space. However, no single record or array can be greater than 16 megabytes.

Modules that are link-edited together must also reside in the same address space (either above or below the 16-megabyte address line).

To take advantage of 31-bit addressing, you must either:

- Link-edit your program using the XA option in the PASCMOD CLIST
- Use the AMODE and RMODE options when link-editing your program with the linkage editor.

Every program that executes in MVS/XA or MVS/ESA is assigned two attributes: AMODE (addressing mode) and RMODE (residence mode). AMODE is the attribute of an entry point into a load module that specifies the addressing mode in effect when the load module is entered at run-time. RMODE is the attribute of a load module that specifies the residence mode of a load module when it is loaded into virtual storage for execution.

Valid AMODE and RMODE specifications are shown in Figure 9.

Attribute	Meaning
AMODE=24	24-bit addressing mode
AMODE=31	31-bit addressing mode
AMODE=ANY	Either 24-bit or 31-bit addressing mode
RMODE=24	The module must reside in virtual storage below 16 megabytes. Use RMODE=24 for 31-bit programs that have 24-bit dependencies.
RMODE=ANY	Indicates that the module can reside anywhere in storage.

Figure 9. AMODE and RMODE Specifications

For more information on AMODE and RMODE, see *MVS/Extended Architecture Linkage Editor and Loader User's Guide*.

Sample TSO Session to Invoke 31-Bit Addressing Mode

Figure 10 shows how to compile an existing source module, link-edit, and execute in 31-bit addressing mode. The commands you enter from the terminal are preceded with =>.

```
    READY

 => VSPASCAL LANDER SYSPRINT(A) LIST

    INVOKING VS PASCAL RELEASE 2.0
     STARTING LANGUAGE ANALYSIS PASS...
     STARTING OPTIMIZATION PASS...
     STARTING CODE GENERATION PASS...
     NO COMPILER DETECTED ERRORS
    SOURCE LINES:  47; COMPILE TIME:  0.19 SECONDS; COMPILE RATE: 15032

    READY

 => PASCMOD LANDER LOAD(PROGRAMS(LANDER)) XA
    READY

 => ALLOC DDNAME(INPUT) DSNAME(*)
    READY

 => ALLOC DDNAME(OUTPUT) DSNAME(*)
    READY

 => CALL PROGRAMS(LANDER) 'PARMS GO HERE'
```

Figure 10. Compiling, Link-Editing, and Running a Program under MVS/XA

Chapter 3. How to Run a Program in an MVS Batch Environment

This chapter describes how to compile and execute VS Pascal programs in an MVS batch environment using the IBM-supplied cataloged procedures. If you are not using VS Pascal under MVS batch mode, you can skip this chapter.

VS Pascal requires a minimum region size of 512K to compile a program under MVS. A compiled and link-edited program can execute in a 448K region.

Job Control Language

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

The JCL statements essential to run a VS Pascal job are:

- JOB statement, which identifies the start of the job.
- EXEC statement, which identifies a job step and the program to be executed, either directly or by means of a cataloged procedure.
- 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 list of publications describing JCL is given in the "Bibliography" on page 295.

Compiling a Program that Uses the %INCLUDE Compiler Directive

The %INCLUDE directive in a source program identifies a member from a partitioned data set and directs the compiler to search for that member. When found, that data set member becomes the input stream for the compiler. After the compiler reads the entire member, it returns to the previous input stream, at the point immediately after the %INCLUDE directive.

At compile time, you must use the SYSLIB DD statement in procedure step PASC to tell the compiler which data sets it must search for members named by a %INCLUDE. Remember, also, that those data sets must be identified to the operating system (using standard JCL) before the program can be compiled.

The order in which data sets are identified using the SYSLIB DD statement can affect program results. To understand why, remember that the %INCLUDE directive can identify a data set member in one of two ways:

- *By naming both the data set and the member.* In this case, the compiler searches only the data set named.

- *By naming only the member*. In this case, the compiler searches the default library, SYSLIB, which contains all the data sets *in the order that you listed them* with the SYSLIB DD statement. When a %INCLUDE identifies a member name that exists in two or more data sets, the compiler uses the first so-named member that it encounters.

As an example, suppose you use SYSLIB to list, in order, LIB1, LIB2, and LIB3. Both LIB2 and LIB3 contain a member named CONSTANTS. When the compiler encounters a %INCLUDE CONSTANTS (without an accompanying data set name), it first searches LIB1. Not finding CONSTANTS in LIB1, it goes to LIB2, where it finds and uses CONSTANTS. If the program must use "constants" from LIB3 instead, you must either (1) change the %INCLUDE directive in the source to specify both the data set name *and* member name or (2) list LIB3 before LIB2 with the SYSLIB DD statement.

You can specify an include library with the INCLLIB parameter of the cataloged procedures. You can also override the SYSLIB DD statement by specifying a DD statement with the name PASC.SYSLIB. Here is an example.

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

Passing Compile-Time Options

Compile-time options (see Chapter 13, "Compile-Time Options" on page 155) are parameters passed to the compiler specifying whether or not a particular feature is to be active. You can specify a list of compile-time options with the PARM parameter of the EXEC statement.

For example, to compile a program with the Interactive Debugging Tool enabled (DEBUG) and without a cross-reference table (NOXREF), issue the following command:

```
//   EXEC PASCC,PARM='DEBUG,NOXREF'
```

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. The compiler listing is written to the data set specified by the SYSPRINT DD statement. For a detailed description of the information in the source listing, see Chapter 4, "How to Read VS Pascal Listings" on page 33.

Cross-System Compilation

You can compile a VS Pascal program under MVS and link-edit and run the object module under VM.

Using Cataloged Procedures

A regularly used set 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 called 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). A cataloged procedure is included in a job by specifying its name in an EXEC statement instead of the name of a program.

Using 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.

IBM supplies several cataloged procedures for use with the VS Pascal compiler. In most cases, you will use one of these procedures to run a VS Pascal job.

IBM recommends that you review these procedures and modify them to make the most efficient use of your resources and to tailor them to your operating procedures.

The IBM-supplied cataloged procedures are:

PASCC Compile only (step name: PASC)
PASCL Link-edit only (step name: LKED)
PASCG Run only (step name: GO)
PASCCL Compile and link-edit (step names: PASC, LKED)
PASCLG Link-edit and run (step names: LKED, GO)
PASCCG Compile, load, and run (step names: PASC, GO)
PASCCLG Compile, link-edit, and run (step names: PASC, LKED, GO)

See Chapter 12, "MVS Batch Cataloged Procedures" on page 137 for a listing of each procedure.

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 that reads the input data set is PASC. For example, here are the JCL statements that you might use to compile, link-edit, and execute a VS Pascal program:

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

The compiler listings and diagnostics 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 another data set, you must override the ddname SYSLIN in procedure step PASC.

For example, to compile a program stored in data set T123.SORT.PASCAL and store the resulting object module in a data set named T123.SORT.OBJ, use the following JCL:

```
//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)
```

Combining Release 1 and Release 2 Code

When combining units compiled with VS Pascal Release 1 and VS Pascal Release 2, you must link-edit the modules with the VS Pascal Release 2 libraries.

Because some debugger tables have been changed to allow larger programs to be debugged, units compiled with DEBUG under Release 1 must be recompiled using Release 2.

How to Compile a Program Using PASCC

The PASCC cataloged procedure ("PASCC Procedure" on page 139) compiles one VS Pascal source module and produces an object module. It consists of one step, PASC.

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. PASC specifies the term MOD in the DISP parameter, and as a result, if you invoke the procedure PASCC 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.

How to Link-Edit a Program Using PASCL

The PASCL cataloged procedure ("PASCL Procedure" on page 141) link-edits a VS Pascal source module and produces a load module. It consists of one step, LKED.

The DD statement with the name SYSLIB within this step specifies the library, or libraries, from which the linkage editor obtains 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 SYSLMOD DD statement. This statement in the cataloged procedure specifies a new temporary library, &&GOSET.

How to Run a Program Using PASCG

The PASCG cataloged procedure ("PASCG Procedure" on page 143) runs a VS Pascal program. It consists of one step, named GO.

Use this procedure when you already have a load module.

How to Compile and Link-Edit a Program Using PASCCL

The PASCCL cataloged procedure ("PASCCL Procedure" on page 144) compiles a VS Pascal source module to produce an object module and then link-edits the object module 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 obtains 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 SYSLMOD DD statement. This statement in the cataloged procedure specifies a new temporary library, &&GOSET. The load module is placed in &&GOSET and given the member name GO.

Placing the load module in a temporary library defined by the SYSLMOD DD statement assumes 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 SYSLMOD DD statement.

When linking multiple modules, you must supply an entry point. The name of the entry point can be either the name of your main program or the name VSPASCAL. To define an entry point, you must specify a linkage editor ENTRY control card for the linkage editor to process. You can do this with a DD statement named SYSIN for step LKED, which references instream data:

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

Multiple invocations of the PASCC cataloged procedure concatenate object modules. This permits you to compile and link-edit several modules conveniently in one job.

The JCL shown in Figure 11 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 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 VSPASCAL
/*
```

Figure 11. Sample JCL to Perform Multiple Compiles and a Link-Edit

How to Link-Edit and Run a Program Using PASCLG

The PASCLG cataloged procedure ("PASCLG Procedure" on page 147) invokes the linkage editor to form a load module and then executes the load module. You can use this procedure when you already have an object module produced by the VS Pascal compiler. This is especially useful if you have access only to the VS Pascal library and not to the VS Pascal compiler.

How to Compile, Load, and Run a Program Using PASCCG

The PASCCG cataloged procedure ("PASCCG Procedure" on page 149) compiles a VS Pascal source module to produce an object module. In the second step (named GO), the loader processes the object module produced by the compiler and immediately runs the resulting executable program.

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 want external references to be resolved, pass its name in the LKLBDSN operand.

Object modules from previous compilations can 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 must be compiled and 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
```

How to Compile, Link-Edit, and Run a Program Using PASCCLG

The PASCCLG cataloged procedure ("PASCCLG Procedure" on page 152) compiles a source module to form an object module, invokes the linkage editor to form a load module, and then executes the load module.

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

How to Modify the Cataloged Procedures for the DEBUG and TRANLIB Options

You can modify the IBM-supplied cataloged procedures for use with the VS Pascal Interactive Debugging Tool and the transient run-time library.

To modify the procedures, you delete the asterisk (*) from the //* in the appropriate statements for the configuration you want to use.

How to Access Data Sets

Every file variable operated upon in your program must have an associated DD statement for the GO step that executes your program. The ddname 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 VS Pascal declaration below are SYSIN, SYSPRINT, and OUTPUTFI, respectively.

```
VAR
  SYSIN,
  SYSPRINT: TEXT;
  OUTPUTFILE: FILE OF INTEGER;
```

If you use the cataloged procedures, you need not define the file named OUTPUT. The cataloged procedures that execute a VS Pascal program (PASCCG, PASCCLG, and PASCLG) contain a DD statement for OUTPUT. OUTPUT is assigned to the output class where the system messages and compiler listings are produced (SYSOUT=*).

To define a data set to store output, you can override the output DD statement in the cataloged procedure by specifying a GO.OUTPUT DD statement.

You must also include a GO.SYSIN DD statement to provide the necessary terminal input for program input. Terminal output defaults to the output class assigned to SYSOUT unless you specify a data set with a GO.SYSPRINT DD statement.

The DD statements GO.OUTPUT, GO.SYSPRINT, GO.SYSIN, and GO.INPUT override those contained in the GO step of the VS Pascal cataloged procedures. Any or all can be used, but the order in which they are coded is sequence sensitive. If used, GO.OUTPUT must be first, and GO.SYSPRINT must be coded before GO.SYSIN and GO.INPUT.

If the VS Pascal input/output manager attempts to open a data set that has an incomplete data control block (DCB), it assigns default values to the DCB as described in "Data Set DCB Attributes" on page 46. If you prefer not to rely on the defaults, then you should specify the LRECL, BLKSIZE, and RECFM 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).

Note: VS Pascal programs using TERMIN to get input from the terminal in MVS get input from SYSIN in MVS batch. Similarly, programs that use TERMOUT to write to the terminal in MVS write to SYSPRINT in MVS batch.

Examples of Batch Jobs

The first example shows a batch job using the PASCCG procedure. The second example shows a batch job using the PASCC and PASCCLG procedures.

Example of Batch Job Using PASCCG

The job control statements shown in Figure 12 compile and run a VS Pascal program consisting of one module. This program uses only the standard files INPUT and OUTPUT.

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 operating procedures established at your site.

The EXEC statement invokes the IBM-supplied cataloged procedure PASCCG. When the operating system encounters this name, it replaces the EXEC statement with a set of JCL statements previously written and cataloged in a system library. The EXEC statement also passes the LIST compile-time option to VS Pascal.

The cataloged procedure contains two steps:

PASC Invokes the VS Pascal compiler to produce an object module.

GO Invokes the loader to process the object module by loading it into memory and including the appropriate run-time library routines. The resulting program runs immediately.

The DD statement PASC.SYSIN indicates that the program to be processed in procedure step PASC follows immediately in the job. 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 GO.INPUT indicates that the data to be processed by the program (in procedure step GO) follows the GO.INPUT statement.

```
//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 12. Sample Batch Job Using PASCCG

Example of Batch Job Using PASCC and PASCCLG

Figure 13 on page 31 shows an example using the PASCC and PASCCLG cataloged procedures.

The EXEC statements in steps 1 and 2 pass the NOXREF compile-time option to VS Pascal. This option suppresses a cross-reference listing.

Step 1 compiles a program using the PASCC cataloged procedure. Step 2 compiles another program and link-edits and executes both programs using the PASCCLG cataloged procedure.

```
//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 VSPASCAL
/*
//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 13. Sample Batch Job Using PASCC and PASCCLG

31-Bit Addressing Mode

Programs compiled by VS Pascal can execute in the 31-bit addressing mode of the operating system. With 31-bit addressing, you have more freedom to define or reference larger data areas, files, and tables, and to create a larger overall program. Your program and its data are no longer constrained to fit in a 16-megabyte address space. However, no single record or array can be greater than 16 megabytes.

Modules that are link-edited together must also reside in the same address space (either above or below the 16-megabyte address line).

To take advantage of 31-bit addressing, you must link-edit your program using the XA parameter in the cataloged procedures (see Chapter 12, "MVS Batch Cataloged Procedures" on page 137).

Every program that executes in MVS is assigned two attributes: AMODE (addressing mode) and RMODE (residence mode). AMODE is the attribute of an entry point into a load module that specifies the addressing mode in effect when the load module is entered at run-time. RMODE is the attribute of a load module that specifies the residence mode of a load module when it is loaded into virtual storage for execution.

Valid AMODE and RMODE specifications are shown in Figure 14.

Attribute	Meaning
AMODE=24	24-bit addressing mode
AMODE=31	31-bit addressing mode
AMODE=ANY	Either 24-bit or 31-bit addressing mode
RMODE=24	The module must reside in virtual storage below 16 megabytes. Use RMODE=24 for 31-bit programs that have 24-bit dependencies.
RMODE=ANY	Indicates that the module can reside anywhere in storage.

Figure 14. AMODE and RMODE Specifications

For more information on AMODE and RMODE, see *MVS/Extended Architecture Linkage Editor and Loader User's Guide*.

Chapter 4. How to Read VS Pascal Listings

This section describes how to read VS Pascal listings. The listings are:

- Compiler options summary
- Source listing
- Cross-reference listing
- Assembler listing
- External symbol dictionary listing
- Instruction statistics.

Compiler Options Summary

The compiler options summary contains information about the parameters and options in effect when the unit was compiled.

```
1 VS PASCAL RELEASE 2.0              :                    08/09/88 10:50:52              PAGE    1

                                        C O M P I L E R   O P T I O N S

2 PARAMETERS PASSED:  NOXREF

3 OPTIONS IN EFFECT: MARGINS(1,72), SEQ(73,80), NOXREF, LANGLVL(EXTENDED), DDNAME(COMPAT), FLAG(I), STDFLAG(E),
                     LINECOUNT(60), PAGEWIDTH(128), CHECK, GOSTMT, HEADER , OPTIMIZE, PXREF, SOURCE, CONDPARM()
```

Figure 15. Sample Compiler Options Summary

1 Page Heading
The first line begins with "VS PASCAL RELEASE 2.0". This line lists information in the following order.

1. The date and time when the program was compiled.
2. The page number.

The second line identifies this page as part of the options summary.

2 Parameter List
The parameter list summarizes the parameters that were passed to VS Pascal.

3 Option List
The option list summarizes the options that were in effect for the compilation.

Source Listing

The source listing contains information about the source program, including nesting and cross-reference information. See Figure 16 for a sample source listing.

```
1 VS PASCAL RELEASE 2.0        SRCHTREE:                              08/09/88 10:50:52                PAGE    3

2 B P C I STMT NO 3                          S O U R C E   P R O G R A M                          PAGE XREF 4
                              INCLUDE 1 FROM SYSLIB(GLOBALS)
                    V---+----1----+----2----+----3----+----4----+----5----+----6----+----7-V SEQ NO
          1:    :                                                                          00000001 *
          1:    :type                                                                      00000002 R
          1:    :  NAMEPTR = @NAMEREC;                                                     00000003 * *
          1:    :  NAMEREC =                                                               00000004 *
          1:    :      record                                                              00000005 R
          1:    :        NAME      : STRING(30);                                           00000006 * 1
          1:    :        LEFT_LINK,                                                        00000007 *
          1:    :        RIGHT_LINK: NAMEPTR;                                            00000008 * 3
          1:    :      end;                                                                00000009 R
          1:    :                                                                          00000010
          1:    :def                                                                       00000011 R
          1:    :  TREETOP : NAMEPTR;                                                     00000012 * 3
                :                                                                        00000004
                :static                                                                  00000005 R
                :  NOTICE : STRING(80);                                                  00000006 * 1
                :                                                                        00000007
                :value                                                                   00000008 R
                :  NOTICE := 'Copyright Columbus Software Corporation 1492';             00000009 3
                :                                                                        00000010
   1            :procedure SEARCH(                                                       00000011 R *
   1            :      const ID: STRING;                                                 00000012 R * 1
   1            :      var  PTR: NAMEPTR);                                               00000013 R * 3
                :EXTERNAL;                                                               00000014 *
                :                                                                        00000015
   1            :procedure SEARCH;                                                       00000016 R 3
   1            :var                                                                     00000017 R
   1            :  LPTR = NAMEPTR;                                                       00000018 * 3
==========ERROR=>      +17
   1            :begin                                                                   00000019 R
   1          1 :  PTR := nil;                                                           00000020 3 1
   1          2 :  LPTR := TREETOP;                                                      00000021 3 3
   1   1      3 :  while LPTR <> nil do                                                  00000022 R 3 1 R
   1   1        :    begin                                                               00000023 R
 1 1   1      4 :      with LPTR@ do                                                     00000024 R 3 R
 1 1 1 1      5 :        if NAME = ID then                                               00000025 R 3 3 R
 1 1 1 1        :          begin                                                         00000026 R
 2 1 1 1      6 :            PTR := LPTR                                                 00000027 3 3
 2 1 1 1      7 :            return                                                      00000028 R
==========ERROR=>               +8
 1 1 1 1        :          end                                                           00000029 R
```

Figure 16 (Part 1 of 2). Sample Source Listing

```
 1 1 1 1        :        else                                                  00000030 R
 1 1 2 1      8 :          if ID < NAME then                                   00000031 R 3 3 R
 1 1 2 1      9 :            LPTR := LEFT_LINK                                 00000032 3 3
*1 1 2 1        :          else                                                00000033 R
 1 1 2 1     10 :            LPTR := RIGHT_LINK                              00000034 3 3
   1   1        :    end (* while *)                                          00000035 R
               :end;.                                                        00000036 R

5 NUMBER OF ERRORS REPORTED:  2
 ERROR   8:  Semicolon ";" expected
 ERROR  17:  Colon ":" expected

6 SOURCE LINES:    48;     COMPILE TIME:   0.06 SECONDS;     COMPILE RATE:  48000 LPM
```

Figure 16 (Part 2 of 2). Sample Source Listing

1 Page Heading

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, the line will be blank.

The second line begins with "VS PASCAL RELEASE 2.0". 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 or function definition which was being compiled when the page boundary occurred.
3. The date and time when the program was compiled.
4. The page number.

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

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
```

B indicates the depth of BEGIN block nesting.

P indicates the depth of procedure nesting.

C indicates the nesting of conditional statements (IF and CASE).

I indicates the nesting of iterative statements (FOR, REPEAT, and WHILE).

3 Statement Numbering

VS Pascal numbers the statements of a routine. STMT NO is the heading of a column that numbers the executable statements of each routine. If the source line originated from an include library, the include number and a colon (":") precede the statement number. These numbers are referenced when a run-time error occurs (see "Reading a VS Pascal Trace-Back Report" on page 73) and when breakpoints are specified in the Interactive Debugging Tool (see Chapter 15, "Interactive Debugging Tool Commands" on page 173).

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

A pair of BEGIN and END statements is not numbered because it serves only as a bracket for a sequence of statements and has no executable code associated with it.

4 Page Cross-Reference Field
If the PXREF compile-time 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.
- An "*" indicates that the corresponding identifier is being declared.
- A "P" indicates that the corresponding identifier is predefined.
- An "R" indicates that the corresponding identifier is a reserved 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.

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.

Note: Message inserts will appear as <???>.

6 Compilation Statistics
The compiler prints summary statistics that tell the number of lines compiled, the time required, and compilation rate in lines per minute of (virtual) processor time.

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

Cross-Reference Listing

The cross-reference listing lists in alphabetic order every identifier used in the unit, giving its attributes and both the page number and the source line number of each reference.

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

Figure 17 shows a sample cross-reference listing.

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

        INCLUDE 1 CAME FROM MEMBER GLOBALS
  1                 2             3
IDENTIFIER        DEFINITION    ATTRIBUTES                                   FORMAT: <PAGE NO>/<INCLUDE NO>:<LINE NO>

ID                  3/12        IN SEARCH, CLASS = CONST PARAMETER, TYPE = STRING, OFFSET = 144
                                  3/25        3/31

LEFT_LINK           3/1:7       IN NAMEREC, CLASS = FIELD, TYPE = POINTER, OFFSET = 32, LENGTH = 4
                                  3/32

LPTR                3/18        IN SEARCH, CLASS = LOCAL VARIABLE, TYPE = POINTER, OFFSET = 152, LENGTH = 4
                                  3/21        3/22        3/24        3/27        3/32        3/34

NAME                3/1:6       IN NAMEREC, CLASS = FIELD, TYPE = STRING, OFFSET = 0, LENGTH = 32
                                  3/25        3/31

NAMEPTR             3/1:3       CLASS = TYPE, TYPE = POINTER, LENGTH = 4
                                  3/1:8       3/1:12      3/13        3/18

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

NIL               PREDEFINED    CLASS = CONSTANT, TYPE = POINTER
                                  3/20        3/22

NOTICE              3/6         CLASS = STATIC VARIABLE, TYPE = STRING, OFFSET = 0, LENGTH = 82
                                  3/9

PTR                 3/13        IN SEARCH, CLASS = VAR PARAMETER, TYPE = POINTER, OFFSET = 148, LENGTH = 4
                                  3/20        3/27

RIGHT_LINK          3/1:8       IN NAMEREC, CLASS = FIELD, TYPE = POINTER, OFFSET = 36, LENGTH = 4
                                  3/34

SEARCH              3/16        CLASS = ENTRY PROCEDURE
                                  3/16

STRING            PREDEFINED    CLASS = TYPE, TYPE = STRING
                                  3/1:6       3/6         3/12

TREETOP             3/1:12      CLASS = DEF VARIABLE, TYPE = POINTER, LENGTH = 4
                                  3/21


NO LANGUAGE ANALYSIS ERRORS DETECTED

SOURCE LINES:    40; COMPILE TIME:   0.07 SECONDS; COMPILE RATE:   34286 LPM


NO OPTIMIZER ERRORS DETECTED

SOURCE LINES:    40; COMPILE TIME:   0.01 SECONDS; COMPILE RATE:   24000 LPM
```

Figure 17. Sample Cross-Reference Listing

1 Identifier
Lists the identifiers used in the unit in alphabetic order.

2 Definition

The definition is of the form: *page/include:line*. *page* is the page number, *include* is the number of the include-member if the reference took place within the member, and *line* is the line number within the program or include-member at which the reference occurred. This column will state PREDEFINED if the identifier is predefined, or CONDPARM if the identifier appeared on a %WHEN directive or in the CONDPARM compile-time option. The reference immediately following the identifier is the place in the source program where the identifier was declared.

3 Attributes

There are six possible attributes:

IN *name*

If the identifier is a record field, 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. *Class* can be:

CONSTANT
A declared constant

CONST PARAMETER
A pass-by-CONST parameter

DEF VARIABLE
A DEF external variable

ENTRY FUNCTION
A function declared as an external entry point

ENTRY PROCEDURE
A procedure declared as an external entry point

EXTERNAL FUNCTION
An external function

EXTERNAL PROCEDURE
An external procedure

FIELD
A record field

FORMAL FUNCTION
A function passed as a parameter

FORMAL PROCEDURE
A procedure passed as a parameter

FORTRAN FUNCTION
An external FORTRAN function

FORTRAN SUBROUTINE
An external FORTRAN subroutine

FUNCTION
A user-defined or standard function

GENERIC PROCEDURE
A generic procedure

LABEL
A statement label

LOCAL VARIABLE
An automatic variable

MAIN ENTRY POINT
A MAIN procedure

PROCEDURE
A user-defined or standard procedure

REENTRANT ENTRY POINT
A REENTRANT procedure

REF VARIABLE
An REF external variable

STATIC VARIABLE
A static variable

TYPE
A type identifier

VALUE PARAMETER
A pass-by-value parameter

VAR PARAMETER
A pass-by-VAR parameter

UNDECLARED
An undeclared identifier

TYPE = *type*
This attribute gives the type of the identifier. *Type* can be:

ARRAY
An array type

BOOLEAN
A Boolean type

CHAR
A character type

FILE
A file type

GCHAR
A graphic character type

GSTRING
A graphic string type

INTEGER
A fixed-point numeric

POINTER
A pointer type

REAL
A floating-point numeric

RECORD
A record type

SCALAR
An 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 notation) of an automatic variable or parameter within a routine's dynamic storage area (DSA); the displacement of a record field within the associated record; or the offset of a static variable in the static area.

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.

Assembler Listing

The compiler produces a pseudo-assembler listing of your program if you specify the LIST option. Figure 18 shows a sample assembler listing.

```
 1       2                3                4
 LOC   OBJECT CODE        STMT    P S E U D O   A S S E M B L Y   L I S T I N G

                                  *        if NAME = ID then
0000DA  18E3                      56       LR    14,03
0000DC  48F0  E000                57       LH    15,0(,14)
0000E0  41E0  E002                58       LA    14,2(,14)
0000E4  5840  D090                59       L     04,144(,13)
0000E8  4850  4000                60       LH    05,0(,04)
0000EC  5650  ****                61       O     05,=X'40000000'
0000F0  4140  4002                62       LA    04,2(,04)
0000F4  0FE4                      63       CLCL  14,04
0000F6  4770  ****                64       BNE   @4L3
                                  *          begin
                                  *            PTR := LPTR;
0000FA  5840  D094                67       L     04,148(,13)
0000FE  5030  4000                68       ST    03,0(,04)
                                  *            return
000102  47F0  ****                70       B     @4L2
000106                            71  @4L3 DS    0H
                                  *          end
                                  *        else
                                  *          if ID < NAME then
000106  58E0  D090                75       L     14,144(,13)
```

Figure 18 (Part 1 of 2). Sample Assembler Listing

```
00010A  48F0  E000           76          LH    15,0(,14)
00010E  41E0  E002           77          LA    14,2(,14)
000112  5830  D09C           78          L     03,156(,13)
000116  1843                 79          LR    04,03
000118  4850  4000           80          LH    05,0(,04)
00011C  5650  ****           81          O     05,=X'40000000'
000120  4140  4002           82          LA    04,2(,04)
000124  0FE4                 83          CLCL  14,04
000126  47B0  ****           84          BNL   @4L6
                                 *           LPTR := LEFT_LINK
00012A  D203  D098  3020      86          MVC   152(4,13),32(03)
000130  47F0  ****           87          B     @4L7
000134                       88  @4L6    DS    0H
                                 *         else
                                 *           LPTR := RIGHT_LINK
000134  5830  D09C           91          L     03,156(,13)

000138  D203  D098  3024      92          MVC   152(4,13),36(03)
00013E                       93  @4L7    DS    0H
00013E  47F0  2028           94          B     @4L1
000142                       95  @4L2    DS    0H
000142  D203  C028  D000      96          MVC   40(4,12),0(13)
000148  58D0  D004           97          L     13,4(,13)
00014C  98E5  D00C           98          LM    14,05,12(13)
000150  07FE                 99          BR    14
000154  0000  0000          100                =F'0'
000158  4000  0000          101                =X'40000000'
00015C  0000  0000          102                =V(TREETOP)
```

Figure 18 (Part 2 of 2). Sample Assembler Listing

1 Location Information
Location relative to the beginning of the unit in bytes (hexadecimal).

2 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 ("****").

3 Statement Information
Statement number in listing.

4 Language Description
Basic assembler language description of generated instruction.

Note: Intermixed with the assembler instructions are the source lines from which the instructions were generated. The source lines appear as comments in the listing.

External Symbol Dictionary Listing

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 linkage editor/loader to resolve inter-module linkages.
Figure 19 shows a sample external symbol dictionary listing.

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

 1           2   3   4       5
NAME         TYPE  ID  ADDR    LENGTH          NAME         TYPE  ID  ADDR    LENGTH

SRCHTREE     SD     1 000000   000290          SEARCH       LD     0 00009A   000001
@STATIC      PC     2 000000   000052          TREETOP      CM     3 000000   000004
AMPXSEGE     ER     4 000000
```

Figure 19. Sample ESD Table

1 Name
Name of the symbol.

2 Type
The classification of the symbol:

SD	Symbol definition (corresponds to name of module)
LD	Local definition (entry routines)
ER	External reference (external routines)
CM	Common (corresponds to DEF variables)
PC	Private code (where static variables are located)

3 ID
Number provided to the loader in order to relocate address constants correctly.

4 ADDR
Offset (in hexadecimal) in the CSECT of an LD entry.

5 Length
Size in bytes (in hexadecimal) of an SD, CM, or PC entry.

Instruction Statistics

If VS Pascal is requested to produce an assembler listing, it will also summarize the usage of System/370 instructions generated by the compiler. The table is sorted by frequency of occurrence. Figure 20 shows a sample instruction statistics table.

```
                        I B M / 3 7 0   I N S T R U C T I O N   U S A G E
                  1      2   3
          OP CODE  NUM    %            OP CODE  NUM   %            OP CODE  NUM   %            OP CODE  NUM   %

            L       10 21.74             ST       3  6.52           BNE      1  2.17           CLC      1  2.17
            LA       4  8.70             LR       2  4.35           BNL      1  2.17           BC       1  2.17
            LH       4  8.70             CLCL     2  4.35           LM       1  2.17           SR       1  2.17
            MVC      4  8.70             O        2  4.35           BAL      1  2.17           BE       1  2.17
            B        3  6.52             BR       2  4.35           LTR      1  2.17           STM      1  2.17

NO CODE GENERATION ERRORS DETECTED

SOURCE LINES:    48; TRANSLATE TIME:   0.11 SECONDS; TRANSLATE RATE:  26182 LPM

                     TOTAL TIME:   0.21 SECONDS;     TOTAL RATE:  13714 LPM
```

Figure 20. Sample Instruction Statistics Table

1 OP CODE
The assembler mnemonic of the instruction.

2 NUM
The number of times the instruction was generated.

3 %
The percentage (of total instructions) that a given instruction was generated.

Chapter 5. How to Use the Input/Output Facilities

An essential part of every program you develop is the reading and writing of data. Your program retrieves information, processes it as you specify, and then produces the results you want.

The source of the information and the display target for the output can be one or more of the following:

- A direct-access device
- A magnetic tape
- A printer
- A terminal
- A card reader or punch
- Another program to which you pass data.

VS Pascal uses input and output statements to read and write data. An input statement allows you to read data from a device. An output statement allows you to write data to a device.

Input and output statements operate on files. A file is a collection of physical and logical records; that is, a sequence of pieces of information that your program can process. Under MVS, a file is also known as a data set.

VS Pascal uses the MVS or VM access methods to read and write data. The access methods VS Pascal uses are:

- Queued sequential access method (QSAM)
- Basic partitioned access method (BPAM)
- Basic direct access method (BDAM).

Records in a data set processed by QSAM are stored and retrieved as logical records. QSAM handles any physical blocking or deblocking required. On input, QSAM anticipates the need for a record based on its physical order; normally, the desired record is in storage, ready for use, before the request for it is made. On output, QSAM holds the logical records in a buffer and performs physical output only when the buffer is filled. QSAM is used for sequential data sets.

With BPAM, a data set consists of a number of members and a directory that holds the name and location of each member. A data set organized in this manner is called a partitioned data set (PDS) under MVS or MACLIB under VM. BPAM maintains and accesses the directory; once BPAM locates the desired member, the records within the member are processed.

With BDAM, records may be organized in any manner. The data set must reside on a direct-access device. Records are stored and retrieved by actual addresses within the data set.

VS Pascal associates a file variable with a data set by means of a ddname. Under VM, you would use the FILEDEF command to associate a file variable with a data set. Under MVS/TSO, you would use the TSO ALLOCATION command to associate the file variable with a data set. Under MVS batch mode, you would use the DD statement to associate a file variable with a data set.

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 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 "Options for Opening a File" on page 54).

Once a file is associated with a ddname, the only way to change that association is to reopen the file specifying a new ddname.

You should explicitly specify ddnames for files which are elements of arrays, fields of records, or pointer qualified. If the ddname is not explicitly specified for such files, VS Pascal will generate a ddname. See the DDNAME option in "Options for Opening a File" on page 54.

You can also specify how VS Pascal generates ddnames for files with the DDNAME compile-time option. Use DDNAME(COMPAT) if you want VS Pascal to use the rules for Pascal/VS Release 2.2 to generate ddnames. Use DDNAME(UNIQUE) if you want VS Pascal to use unique ddnames for each file to conform to standard Pascal scoping rules. See Figure 27 on page 55.

Data Set DCB Attributes

Associated with every VS Pascal file variable at run-time is a Data Control Block (DCB), containing 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).

VS Pascal supports all of the record formats supported by QSAM, such as fixed, variable, undefined, fixed-blocked, variable-blocked, and so on.

A VS Pascal 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 these files will be prefixed with the appropriate control character. Thus, the first character position of each record is not directly accessible from the VS Pascal program. If the NOCC option is specified when the file is opened, no control character will be prefixed and the first character *is* accessible (see "Options for Opening a File" on page 54). For input files, VS Pascal will always get the first character, even if it is a carriage control character.

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

- By specifying them in the associated data set definition (in CMS through the FILEDEF command, in TSO the ALLOC/ATTR commands, in MVS batch mode the DD statement)
- By specifying them when the file is opened by means of the options string. (See "Options for Opening a File" on page 54.)

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

For TEXT files, if you do not specify LRECL, BLKSIZE, or RECFM, then the following defaults will apply:

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

For record files, if you do not specify LRECL, BLKSIZE, or RECFM, then the following defaults will apply:

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

If you specify some of the attributes, then the defaults will be applied using the following criteria:

- RECFM = V is used for TEXT files.
- RECFM = F is used for record files.
- If RECFM = F, then the BLKSIZE is to be equal to the LRECL or to be a multiple thereof.
- If RECFM = V, then the BLKSIZE is to be at least four bytes greater than the LRECL.

Types of Files

Text Files

VS Pascal supports both fixed-length and variable-length record formats for TEXT files. Characters are stored in the EBCDIC character set.

The predefined data type TEXT is used to declare a TEXT file variable in VS Pascal. The pointer associated with each file variable points to positions within a physical I/O buffer.

Record Files

The logical record length (LRECL) of a file must be at least large enough to contain the file's base component; otherwise, a run-time error message 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 file's component type, then the extra space in each logical record is wasted.

Opening a File

Before you can read data from a file or write data to a file, you must open the file. The following sections describe ways of opening a file depending on how you want to process that file.

Opening a File for Input (RESET)

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

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

Where	Represents
f	A file variable.
options	An optional string that contains the open options (see "Options for Opening a File" on page 54).

Normally, RESET allocates a buffer, reads the first logical record of the file into the buffer, and positions the file pointer at the beginning of the buffer. Therefore, given a TEXT file F, the execution of the statement RESET(F) would imply that "F@" would reference the first character of the file.

If a RESET operation is performed on an open file, the file is closed and then reopened.

Figure 21 shows an example of using the RESET procedure on a TEXT file.

```
PROGRAM EXAMPLE;
VAR
  SYSIN  : TEXT;
  C      : CHAR;
BEGIN
  (*open SYSIN for input    *)
  RESET(SYSIN,'NAME=SYSIN.TEXT');
  (*get first character of file*)
  C := SYSIN@;
END.
```

Figure 21. Example of Using the RESET Procedure on a TEXT File

Opening a File for Interactive Input

Because 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 supposed to write out prompting messages before 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 first file operation is performed (namely a GET or READ). If the file is a TEXT file, the end-of-line condition (see "End-of-Line Condition" on page 64) is initially set to TRUE.

Figure 22 shows an example of opening a file for interactive input.

```
PROGRAM EXAMPLE;
VAR
  SYSIN  : TEXT;
  DATA   : STRING(80);
BEGIN
  RESET(SYSIN,'INTERACTIVE');  (* Open SYSIN for interactive input. *)
  WRITELN(' ENTER DATA: ');    (* Prompt for response.              *)
  READLN(SYSIN,DATA);          (* Read in response.                 *)
END.
```

Figure 22. Example of Opening a File for Interactive Input

Opening a File for Output (REWRITE)

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

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

Where	Represents
f	A file variable.
options	An optional string that contains the open options (see "Options for Opening a File" on page 54).

REWRITE positions the file pointer at the beginning of an empty buffer. If the file is already open, it is closed before being reopened.

Figure 23 shows an example of opening a TEXT file with the REWRITE procedure.

```
PROGRAM EXAMPLE;
VAR
  SYSPRINT : TEXT;
BEGIN
  REWRITE(SYSPRINT);
  WRITELN(SYSPRINT,'MESSAGE');
END.
```

Figure 23. Example of Opening a TEXT File with the REWRITE Procedure

Figure 24 shows an example of how to open a record file with the REWRITE procedure.

```
PROGRAM EXAMPLE;
VAR
  OUTFILE : FILE OF INTEGER;
  I       : INTEGER;
BEGIN
  REWRITE(OUTFILE, 'BLKSIZE=1600,LRECL=4,RECFM=F');
  I := 3;
  OUTFILE@ := I;
  PUT(OUTFILE);
END.
```

Figure 24. Example of Opening a Record File with the REWRITE Procedure

Opening a File for Updating (UPDATE)

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

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

Where	Represents
f	A file variable.
options	An optional string that contains the open options (see "Options for Opening a File" on page 54).

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, 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.

Figure 25 shows an example of how to open a record file for updating.

```
PROGRAM EXAMPLE;
VAR
  F    : FILE OF
           RECORD
              NAME: STRING(30);
              AGE : 0..99;
           END;
BEGIN
  UPDATE(F);
  WHILE NOT EOF(F) DO     (* Update each record by incrementing age. *)
   BEGIN
      F@.AGE := F@.AGE + 1;
      PUT(F);
      GET(F);
   END;
END.
```

Figure 25. Example of Opening a Record File for Updating

Opening a Partitioned Data Set For Input (PDSIN)

The PDSIN procedure opens a partitioned data set (PDS) for input. This procedure has the form:

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

Where	Represents
f	A file variable.
options	A string that contains the open options (see "Options for Opening a File" on page 54). If no member name is specified in the options string (using MEMBER=*name*), the member TEMPNAME will be assumed.

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.

Note: All operations that may be applied to "partitioned data sets" under MVS may be applied to MACLIB's and TXTLIB's under CMS. See "Accessing a PDS in a CMS Environment" on page 52 for more information.

See Figure 28 on page 58 for an example of opening a partitioned data set.

Opening a Partitioned Data Set For Output (PDSOUT)

The PDSOUT procedure opens a partitioned data set (PDS) for output. This procedure has the form:

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

Where	Represents
f	A file variable.
options	A string that contains the open options (see "Options for Opening a File" on page 54). If no member name is specified in the options string (using MEMBER=*name*), the member TEMPNAME will be assumed.

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

Note: All operations that may be applied to "partitioned data sets" under MVS may be applied to MACLIB's and TXTLIB's under CMS. See Accessing a PDS in a CMS Environment for more information.

See Figure 28 on page 58 for an example of opening a partitioned data set.

Accessing a PDS in a CMS Environment

In a CMS environment, members of MACLIBs may be accessed as partitioned data sets via the MVS 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 VS Pascal program, the following commands would be run before 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 before executing the VS Pascal 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
```

Opening a File for Terminal Input (TERMIN)

You can use the TERMIN procedure to send input directly to your terminal without going through the normal ddname interface. TERMIN opens a TEXT file for interactive input from your terminal. This procedure has the form:

```
TERMIN(f,options)
```

Where	Represents
f	A file variable.
options	An optional string that contains the open options (see "Options for Opening a File" on page 54).

Notes:

1. The TERMIN procedure opens the file with the INTERACTIVE attribute as described in "Opening a File for Interactive Input" on page 49.
2. The EOF function always returns FALSE for such files, because the "end-of-file condition" does not apply to files opened with the INTERACTIVE attribute.

Figure 26 shows an example of terminal input and output.

Note: Under MVS batch, files opened with TERMIN will read from SYSIN. Input data normally entered from the terminal must be supplied by the JCL for the batch job, and you must provide a DD statement for SYSYIN.

Opening a File for Terminal Output (TERMOUT)

You can use the TERMOUT procedure to send output directly to your terminal without going through the normal ddname interface. TERMOUT opens a TEXT file for terminal output. This procedure has the form:

```
TERMOUT(f,options)
```

Where	Represents
f	A file variable.
options	An optional string that contains the open options (see "Options for Opening a File" on page 54).

Figure 26 shows an example of terminal input and output.

```
PROGRAM EXAMPLE;
VAR
  TTYIN, TTYOUT: TEXT;
        : INTEGER;
BEGIN
  TERMIN(TTYIN);
  TERMOUT(TTYOUT);
  WRITELN(TTYOUT,'ENTER DATA:');
  READLN(TTYIN,I);

END.
```

Figure 26 Example of Terminal Input and Output

Note: Under MVS batch, files opened with TERMOUT will write to SYSPRINT.

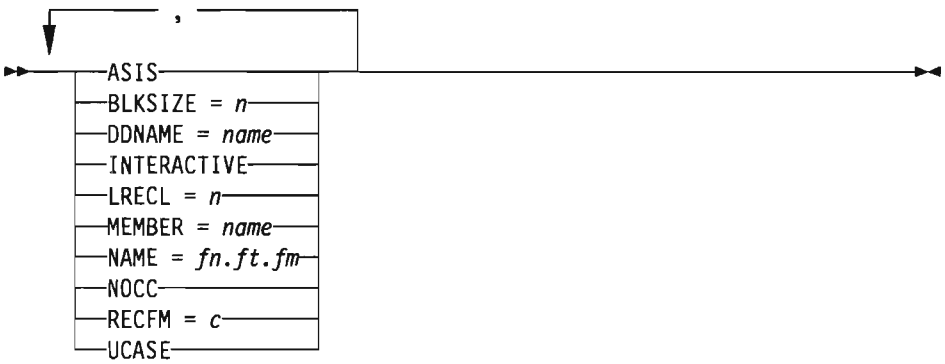
Options for Opening a File

These options are valid for the following I/O routines:

PDSIN
PDSOUT
RESET
REWRITE
TERMIN
TERMOUT
UPDATE

All VS Pascal 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:



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

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

ASIS (CMS only)
ASIS causes text being read from a file to be read as is without translation to upper case (this is the opposite of UCASE).

ASIS applies only to the TERMIN procedure.

BLKSIZE = *n*
BLKSIZE specifies 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.

BLKSIZE applies only to the procedure REWRITE.

DDNAME = *name*
DDNAME 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 you do not specify a DDNAME, VS Pascal generates a ddname according to the guidelines in Figure 27. The ddname will depend on the compile-time option in effect:

- DDNAME(COMPAT) forces VS Pascal to generate ddnames that match the identifiers used in the program. Remember that the operating system does not observe VS Pascal scoping rules. For example, a file variable F in one scope would be different from another file variable F in another scope, but both would have the ddname F. This may cause the first occurrence of F to be overwritten by the second occurrence of F when DDNAME(COMPAT) is in effect. You must make sure that your program contains no duplicate ddnames.
- DDNAME(UNIQUE) instructs VS Pascal to generate unique ddnames based on the names used in the program, thus ensuring that multiple occurrences of a file are not overwritten.

When the File Name Is	And DDNAME(COMPAT) Is In Effect	Or DDNAME(UNIQUE) Is In Effect
• A simple variable.	The initial ddname is the first eight characters of the variable name. Variable names of fewer than eight characters are padded on the right with blanks. For example, for a variable called F, VS Pascal generates the ddname F followed by seven blanks. For a variable called CHECKBOOK, VS Pascal generates the ddname CHECKBOO.	1. **For global variables specified in the program parameter list**, the ddname is the first eight characters of the variable name. 2. **For files opened with TERMIN**, the ddname is $CONSOLE (SYSIN in MVS batch). 3. **For files opened with TERMOUT**, the ddname is $CONSOLE (SYSOUT in MVS batch). 4. **Otherwise**, the initial ddname is the first eight characters of the variable name. Variable names of fewer than eight characters are padded on the right with a pound sign (#). At run time, VS Pascal begins to replace the ddname with digits and a single pad character, starting on the right-hand side of the ddname. For example, if the first file opened is named F, the ddname would be F######1. If the next file opened is named CHECKBOOK, the ddname would be CHECKB#2.

Figure 27 (Part 1 of 2). VS Pascal Generated ddnames

When the File Name Is	And DDNAME(COMPAT) Is In Effect	Or DDNAME(UNIQUE) Is In Effect
• An element of an array. • A field of a record. • Pointer-qualified.	1. **For files opened with TERMIN**, the ddname is $CONSOLE (SYSIN in MVS batch). 2. **For files opened without TERMOUT**, the ddname is $CONSOLE (SYSOUT in MVS batch). 3. **Otherwise**, VS Pascal generates an initial name of PASCAL00. At run time, VS Pascal replaces the name with digits, starting on the right-hand side of the name. For example, the first such file opened would have the ddname PASCAL01, and the 100th file opened would be PASCA100.	1. **For files opened with TERMIN**, the ddname is $CONSOLE (SYSIN in MVS batch). 2. **For files opened with TERMOUT**, the ddname is $CONSOLE (SYSOUT in MVS batch). 3. **Otherwise**, VS Pascal generates an initial name of PASCAL00. At run time, VS Pascal replaces the name with digits, starting on the right-hand side of the name. For example, the first such file opened would have the ddname PASCAL01, and the 100th file opened would be PASCA100.

Figure 27 (Part 2 of 2). VS Pascal Generated ddnames

DDNAME applies to the following procedures: RESET, REWRITE, UPDATE, PDSIN, PDSOUT, TERMIN, and TERMOUT.

INTERACTIVE

INTERACTIVE indicates that the file is to be opened for input as an interactive file.

INTERACTIVE applies to the procedures RESET and PDSIN (and is implied for TERMIN).

LRECL = *n*

LRECL specifies a logical record length to be associated with an output file. The value (indicated by *n*) will override any 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. Thus, if text is being written to such a file, the LRECL must be 4 bytes longer than the longest line to be written.

Note: The 4-byte length descriptor for each record of a V-record file is an MVS convention.

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.

LRECL applies to the procedures REWRITE, TERMIN, and TERMOUT.

MEMBER = *name*

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

MEMBER applies to the procedures PDSIN and PDSOUT. If you do not specify a member name for PDSIN and PDSOUT, VS Pascal uses the member name TEMPNAME.

NAME = *fn.ft.fm* **(CMS only)**

NAME specifies the name of a CMS file to be associated with the file variable. This option has no effect 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 except with the UPDATE procedure.

NAME applies to the following procedures: RESET, REWRITE, UPDATE, PDSIN, and PDSOUT.

NOCC

NOCC causes data to be placed at the first character position of a file, regardless of any carriage control character required in this position by the RECFM.

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.

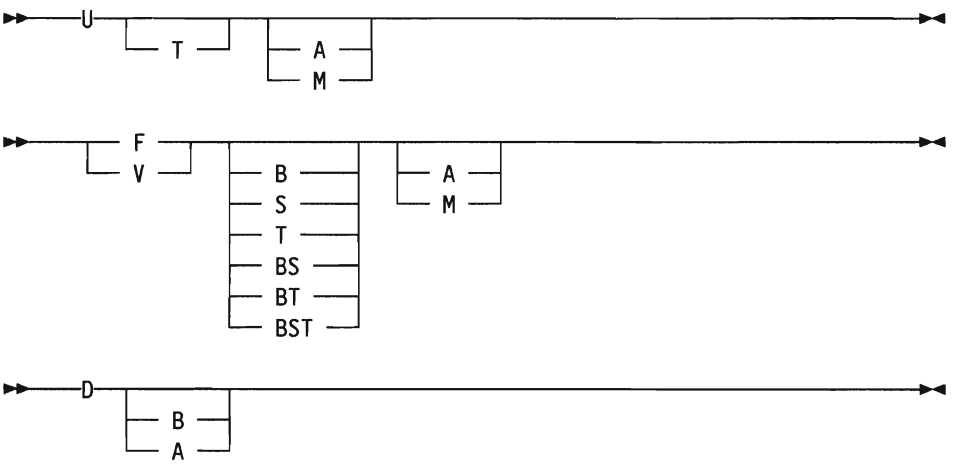
The NOCC option causes such files to be treated as though control characters are not significant; that is, data will be placed within each record at the first character position. This option allows control characters to be generated explicitly.

NOCC applies only to the procedure REWRITE.

RECFM = *c*

RECFM specifies a record format to be associated with an output file. This specification (indicated by *c*) will override a RECFM specification on the ddname definition.

VS Pascal supports all record formats that QSAM supports:



A Specifies that the records in the data set contain ASA print control characters.

B Specifies that the data set contains blocked records.

D Specifies that the data set contains variable-length tape records.

F Specifies that the data set contains fixed-length records.

M Specifies that the records in the data set contain machine print control characters.

S Specifies, for fixed-length records, that the records are to be written as standard blocks. For variable-length and BDAM records, it specifies that the records are to be written as spanned blocks.

T Specifies that track overflow is used with the data set.

U Specifies that the data set contains undefined-length records.

V Specifies that the data set contains variable-length records.

RECFM applies to procedure REWRITE.

UCASE (CMS only)
UCASE causes text being read from a file to be translated to upper case (this is the opposite of ASIS). This option applies only to programs running under CMS; it is ignored otherwise.

UCASE applies only to the procedure TERMIN.

Figure 28 shows how to use the open options.

```
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 first 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 member to   *)
        BEGIN                   (* file OUTPUT.                      *)
          READLN(PDS,BUF);
          WRITELN(BUF);
        END;
      READLN(MEMBER);           (* Read next member name.            *)
    END;
END.
```

Figure 28. Example of Using the Open Options

Processing a TEXT File

This section describes how to read data from and write data to a TEXT file.

Reading Data from a TEXT File (GET)

The GET procedure allows you to read data from a file. A call to the procedure has the form:

```
GET(f)
```

Where	Represents
f	A file variable.

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 64) and the file pointer will be positioned to a blank. If, before 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, before 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" on page 65.)

If you attempt to use GET on a TEXT file that has not been opened, it will be implicitly opened for input (as if RESET had been called). This means that the first character of the file will be skipped and the file pointer will be at the second character.

Figure 29 shows how to use the GET procedure on a TEXT file.

```
PROGRAM EXAMPLE;
VAR
  INFILE   : TEXT;
  C1,C2    : CHAR;
  .
  .
BEGIN
  RESET(INFILE);          (* Get first char of file.   *)
  C1 := INFILE@;
  GET(INFILE);            (* Get second char of file.  *)
  C2 := INFILE@;
  .
  .
END.
```

Figure 29. Example of Using the GET Procedure on a TEXT File

Writing Data to a TEXT File (PUT)

The PUT procedure allows you to write data to a file. A call to this procedure has the form:

```
PUT(f)
```

Where	Represents
f	A file variable.

You must open the file for output or update before calling PUT; otherwise, a run-time error message will be issued. Before issuing a PUT operation, the associated output buffer must contain the data to be written.

The PUT procedure, when applied to a TEXT file opened for output, causes the file pointer to be incremented by one character position. If, before 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).

Figure 30 shows how to use the PUT procedure for a TEXT file.

```
PROGRAM EXAMPLE;
VAR
  OUTFILE  : TEXT;
  C        : CHAR;
BEGIN
  REWRITE(OUTFILE);
  OUTFILE@ := C;
  (*Write out value of C*)
  PUT(OUTFILE);
END.
```

Figure 30. Example of Using the PUT Procedure for a TEXT File

Reading Data from a TEXT File (READ)

The READ procedure reads data from a TEXT file.

A call to READ has the forms:

```
READ(f,v)
```

or

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

Where	Represents
f	A file variable. The file variable *f* may be omitted, in which case the file INPUT is assumed.
v	A variable that must be one of the following types. • CHAR (or a subrange thereof) • DBCS fixed string • GCHAR • GSTRING • INTEGER (or a subrange thereof) • REAL (or SHORTREAL) • SBCS fixed string • STRING
n	An optional field length (an integer expression).

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 interactively. (See "Opening a File for Interactive Input" on page 49.)

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

Reading Data from a TEXT File (READLN)

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 READLN 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 before 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.

Figure 31 shows how to use the READLN procedure.

```
PROGRAM COPY;
VAR
  INFILE,
  OUTFILE : TEXT;
  BUF     : STRING(100);
BEGIN
  RESET(INFILE);
  REWRITE(OUTFILE);
  WHILE NOT EOF(INFILE) DO
    BEGIN
      READLN(INFILE,BUF);    (* Ignore characters after column 100  *)
      WRITELN(OUTFILE,BUF);  (* in each line.                       *)
    END;
END.
```

Figure 31. Example of Using the READLN Procedure

Writing Data to a 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	Represents
f	A file variable. The file variable *f* can be omitted, in which case, the file OUTPUT is assumed.
e	An expression that must be of one of the following types: • BOOLEAN • CHAR (or a subrange thereof) • DBCS fixed string • GCHAR • GSTRING • INTEGER (or a subrange thereof) • REAL (or SHORTREAL) • SBCS fixed string • STRING
n, n1, n2	Optional field lengths (integer expressions).

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 longer than the logical record length (LRECL) of the TEXT file, a run-time error message will be generated.

Writing Data to a TEXT File (WRITELN)

A call to WRITELN has the same form as a call to WRITE and performs the same function except that as it causes the current logical record being produced to be completed so that the next output operation will begin a new logical record. The WRITELN procedure is applicable to TEXT files only.

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.

Figure 32 shows how to use the WRITELN procedure.

```
PROGRAM DOUBLESPACE;
VAR
  FILEIN,
  FILEOUT : TEXT;
  BUF     : STRING(255);
BEGIN
  REWRITE(FILEOUT);
  RESET(FILEIN);
  WHILE NOT EOF(FILEIN) DO
    BEGIN
      READLN(FILEIN,BUF);
      WRITELN(FILEOUT,BUF);
      WRITELN(FILEOUT);         (* Insert blank line.     *)
    END;
END.
```

Figure 32. Example of Using the WRITELN Procedure

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	Represents
f	An optional TEXT file variable. The default is OUTPUT.

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

For this procedure to have 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.

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 that contains the hexadecimal value of "8B" in its first character position is written.

Figure 33 shows how to use the PAGE procedure.

```
PROGRAM EXAMPLE;
VAR
  PRINT:  TEXT;
BEGIN
  .
  .
  REWRITE(PRINT);
  .
  .
  PAGE(PRINT);     (* Start new page.     *)
END.
```

Figure 33. Example of Using the PAGE Procedure

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.

Figure 34 shows how to use the EOLN function.

```
PROGRAM EXAMPLE;
VAR
  SYSIN:  TEXT;
  CNT  : 0..32767;
BEGIN
  RESET(SYSIN);               (* Compute length of first logical    *)
  CNT := 0;                   (* record of SYSIN.                   *)
  WHILE NOT EOLN(SYSIN) DO
    BEGIN
      CNT := CNT + 1;
      GET(SYSIN);
    END;
  WRITELN(CNT);
END.
```

Figure 34. Example of Using the EOLN Function

End-of-File Condition

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. Under VM, this can only occur when the file mode is 4.
- 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 or READLN is called and all characters of the last logical record were consumed.

To test for this condition, use the EOF function.

Any calls to GET, READ, or READLN for a file for which the end-of-file condition is true will result in a run-time error.

Figure 35 shows how to use the EOF function on a TEXT file.

```
PROGRAM EXAMPLE;
VAR
  SYSIN:  TEXT;
  CNT  : 0..32767;
BEGIN
  RESET(SYSIN);              (* Compute number of logical records    *)
  CNT := 0;                  (* in file SYSIN.                       *)
  WHILE NOT EOF(SYSIN) DO
    BEGIN
      CNT := CNT + 1;
      READLN(SYSIN);
    END;
  WRITELN(CNT);
END.
```

Figure 35. Example of Using the EOF Function on a TEXT File

Processing a Record File

This section describes how to read data from and write data to a record file.

Reading Data from a Record File (GET)

The GET procedure allows you to read data from a file. A call to the procedure has the form:

```
GET(f)
```

Where	Represents
f	A file variable.

Each call to GET 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.

A record file must be opened for input or update before executing a GET operation; otherwise, a run-time error message will be issued.

Figure 36 shows how to use the GET procedure with a record file.

```
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 36. Example of Using the GET Procedure on a Record File

Writing Data to a Record File (PUT)

The PUT procedure allows you to write data to a file. A call to the procedure has the form:

```
PUT(f)
```

Where	Represents
f	A file variable.

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 produces one logical record.

The file must be opened for output or update before calling PUT; otherwise, a run-time error message will be issued. Before a PUT operation, the associated output buffer must contain the data to be written.

Figure 37 shows how to use the PUT procedure with a record file.

```
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 37. Example of Using the PUT Procedure on a Record File

Reading Data from a Record File (READ)

The statement:

```
READ(f,v)
```

is equivalent to:

```
BEGIN
  V := F@;
  GET(F);
END
```

Where	Represents
f	A record file variable.
v	A variable of the same type as the record file's components.

If file f is not open when READ is called, a run-time error message will be issued.

Writing Data to a Record File (WRITE)

The statement:

```
WRITE(f,e)
```

is equivalent to:

```
BEGIN
  f@ := e;
  PUT(f);
END
```

Where	Represents
f	A record file variable.
e	An expression of the same type as the record file's components.

If file *f* is not open when WRITE is called, a run-time error message will be issued.

Figure 38 shows how to use the READ and WRITE procedures with record files.

```
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 38. Example of Using the READ and WRITE Procedures on Record Files

Accessing a Record File Randomly

VS Pascal permits records of a record file to be accessed in a random order by means of the SEEK procedure. SEEK positions a file pointer to a specific element within a record file.

A call to SEEK has the form:

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

Where	Represents
f	A record file that was previously opened with RESET, REWRITE, or UPDATE.
n	A positive integer expression that corresponds to a record number. 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 MVS batch mode, 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. 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

```
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 non-existent record is being read, CMS will return a buffer of zeroes.

Note: If XTENT is not specified, a default value of 50 is used.

Figure 39 shows how to use the SEEK procedure to access records randomly.

```
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);
  WHILE NOT EOF(IDXFILE) DO    (* Write out names in order of index. *)
    BEGIN
      SEEK(RECFILE,IDXFILE@.RECNO);
      GET(RECFILE);
      WRITELN(OUTPUT,RECFILE@.NAME);
      GET(IDXFILE);
    END;
END.
```

Figure 39. Example of Using the SEEK Procedure

End-of-File Condition

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

- RESET is called for an empty file. Under VM, this can only occur with a file mode of 4.
- The file is opened for output.
- GET or READ is executed for a file in which no more records remain.

To test for this condition, use the EOF function.

Any calls to GET or READ for a file for which the end-of-file condition is true will produce an error message.

Closing a File

The procedure CLOSE closes a file explicitly. A call to this procedure has the form:

```
CLOSE(f)
```

Where	Represents
f	A file variable.

All open files which are declared in the body of a routine are closed implicitly when the routine returns to its invoker. All files which are open when the program terminates will be closed automatically by the VS Pascal run-time environment.

If the variable associated with an open file is destroyed before program termination, the results could be disastrous when VS Pascal attempts to close the file. This problem could occur if the file variable is pointer qualified (exists on the heap) and the heap containing the file is freed with DISPOSEHEAP, or the subheap containing the file is freed with RELEASE. In this case, the file variable must be closed explicitly with a call to CLOSE.

Figure 40 shows how to use the CLOSE procedure.

```
PROGRAM EXAMPLE(OUTPUT);
VAR
  FSTK : ARRAY [1..4] OF TEXT;
  DDNAME : STRING(8);
  I : 1..4;
BEGIN
  DDNAME := 'TEST    ';
  FOR I :=1 TO 4  DO
    BEGIN
      DDNAME[5] := CHR(I + ORD('0'));
      REWRITE(FSTK[I],'DDNAME = '||DDNAME);
      WRITELN(FSTK[I],'Test &numsign',i:1);
      CLOSE  (FSTK[I]);
    END;
END.
```

Figure 40. Example of Using the CLOSE Procedure

Appending Data 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
```

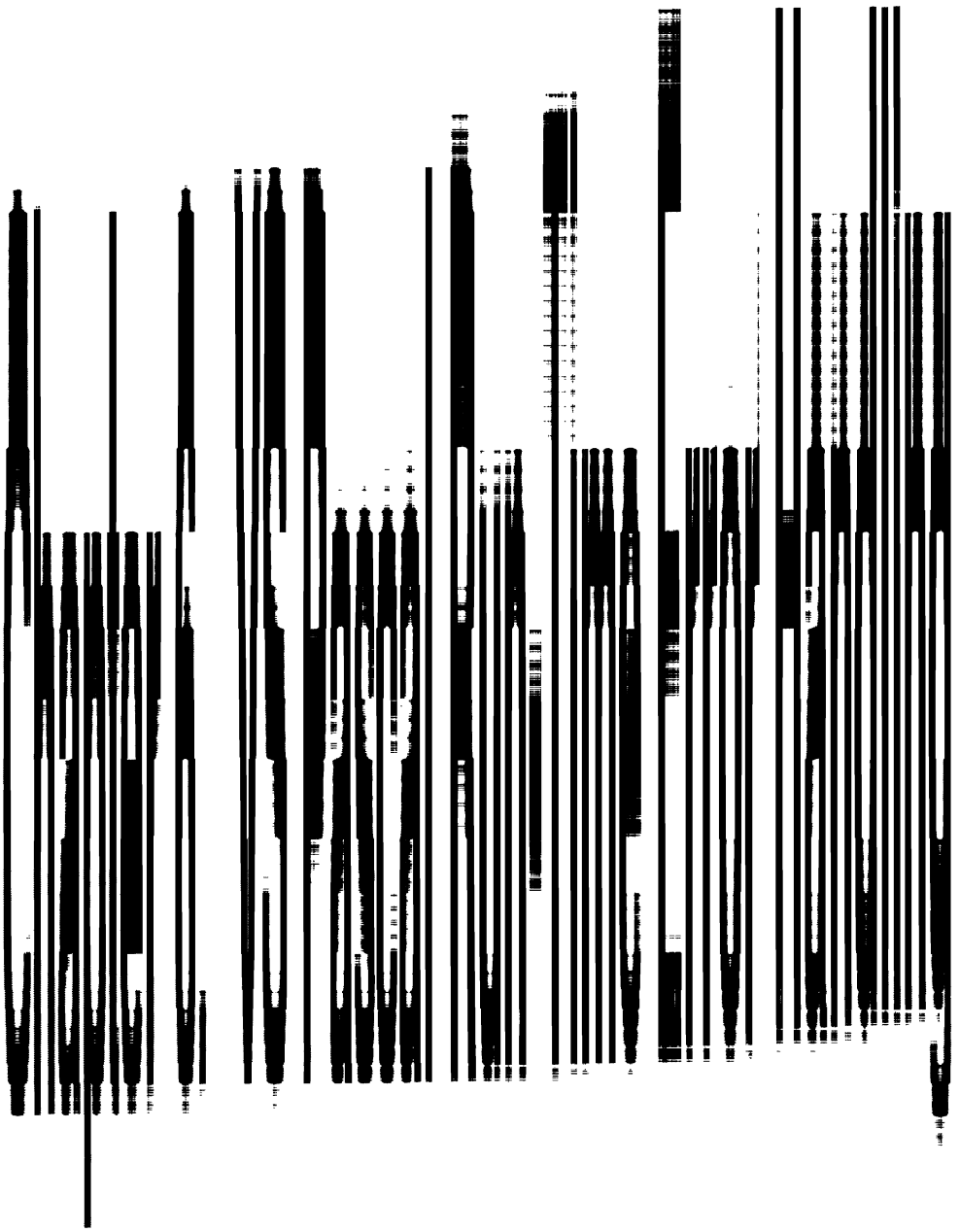
MVS Batch Mode:

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

Unpredictable Actions

VS Pascal will act unpredictably given the following situations:

- Two different files are associated to the same CMS file using the NAME option.
- Two different file variables are open with the same DDNAME. An example of this would be if two file variables have the same first 8 characters (regardless of scoping) when DDNAME(COMPAT) is specified. Another example would be if a recursive procedure opens a file when DDNAME(COMPAT) is specified. See Figure 27 on page 55.



Chapter 6. VS Pascal Run-Time Error Handling

This section explains how to read trace-back reports, how VS Pascal handles run-time errors, how you can produce a routine to control run-time error handling, and how and when VS Pascal produces a symbol dump.

Reading a VS Pascal Trace-Back Report

The VS Pascal trace-back report provides useful information while you are debugging programs. It gives you a list of all the routines in the procedure chain.

For each routine the trace-back report provides:

- The name of the routine
- The statement number of the last statement to be executed in the routine (that is, the statement number of the call to the next routine in the chain)
- The address in storage of the call to the next routine (or in case of an error, the address of the instruction where the error occurred)
- The name of the unit in which the routine is declared.

A VS Pascal trace-back report can be produced in several ways:

- A trace-back report is produced when the predefined TRACE procedure is called in your program, or when you issue a TRACE debugging command. The last line of Figure 41 shows that VS Pascal called the user's main program in the unit 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.

```
     TRACE BACK OF CALLED ROUTINES
Routine           stmt at address in module
HASHKEY             9     0002018C   HASHCSEG
GET_HASH_PTR        2     00021208   HASHBSEG
SEARCH_ID           9     000213C8   HASHBSEG
READ_ID             3     00021550   HASHBSEG
<MAIN-PROGRAM>     24     00020278   HASHASEG
VSPASCAL                  000205FA
```

Figure 41. TRACE Routine Called by a User Program

- VS Pascal produces a trace-back report when a program error occurs. Figure 42 shows a trace-back from a program error. The trace-back contains an error message indicating a fixed-point overflow. The trace-back report identifies the routine and the statement number where the error occurred. The error occurred at statement 3 in routine FACTORIAL on the third recursive call.

```
AMPX018E Fixed Point Overflow
     TRACE BACK OF CALLED ROUTINES
Routine           stmt at address in module
FACTORIAL           3     0002014C   TEST
FACTORIAL           3     0002014C   TEST
FACTORIAL           3     0002014C   TEST
<MAIN-PROGRAM>     17     00020298   TEST
VSPASCAL                  000205FA
```

Figure 42. TRACE Routine Call Due to a Program Error

- VS Pascal produces a trace-back report when a checking error occurs. A checking error occurs when the compiler detects an invalid condition (such as a subscript range error). (For a description of compiler-generated checks, see "CHECK Option" on page 157.) Figure 43 shows a trace-back that occurred from a checking error. The first line of the trace identifies the checking error. The error occurred at statement 4 in routine TRANSLATE.

```
AMPX032E High Bound Checking Error
     TRACE BACK OF CALLED ROUTINES
Routine           stmt at address in module
TRANSLATE           4     00020154   CONVERT
TO_ASCII           10     0002024C   CONVERT
<MAIN-PROGRAM>     17     00020338   CONVERT
VSPASCAL                  000205FA
```

Figure 43. TRACE Routine Call Due to a Checking Error

- VS Pascal produces a trace-back report when an I/O error occurs. Figure 44 shows the trace-back from an I/O error. The trace-back shows that statement 3 of routine INITIALIZE attempted to open a file for which no DDNAME definition exists.

```
AMPX041S File could not be opened: SYSIN
     TRACE BACK OF CALLED ROUTINES
Routine           stmt at address in module
INITIALIZE          3     00020154   COPY
<MAIN-PROGRAM>      2     00020218   COPY
VSPASCAL                  000205FA
```

Figure 44. TRACE Routine Call Due to an I/O Error

Note: While performing optimization, the compiler might move the code that tests for an error condition back several statements. Thus, when a run-time error occurs, the statement number indicated in the trace-back report might be slightly less than the number of the statement that caused the error.

Run-Time Checking Errors

Figure 45 shows the checking errors that can occur in a VS Pascal program at run time.

Error	Description
Low bound	• The value of an array subscript is less than the minimum allowed. • The value of a subrange is less than the minimum allowed.
High bound	• The value of an array subscript is higher than the minimum allowed. • The value of a subrange is higher than the minimum allowed. **Note:** When an unsigned check occurs, you might receive a high bound check when the value was actually too low.
NIL pointer	An attempt was made to reference a variable from a pointer that has the value NIL.
CASE label	The value of a CASE selector does not match any of the specified CASE labels and there is no OTHERWISE clause.
String truncation	In a string-to-string assignment, the source string has more characters than the maximum allowed length of the target string.
Assertion failure	An ASSERT statement evaluated to FALSE.
String subscript out of bounds	An indexing operation on a string was greater than the current length of the string.
Function value	A function routine returned to its invoker without being assigned a result.

Figure 45. Run-Time Checking Errors

How VS Pascal Handles Run-Time Errors

VS Pascal detects many kinds of errors during program execution; upon detecting an error, the VS Pascal run-time library provides error handling.

In VM/CMS or MVS/TSO, the output is sent to your terminal (unless you select another ddname with the ERRFILE run-time option).

In MVS batch, the output is sent to SYSPRINT (unless you select another ddname with the ERRFILE run-time option).

The run-time library considers certain errors fatal. Examples of fatal errors are operation exceptions and protection exceptions. When a fatal error occurs, VS Pascal:

1. Issues a message describing the error
2. Displays a trace-back report
3. Terminates program execution.

Other errors, such as checking errors, do not stop program execution. You must determine the extent to which the nonfatal errors affect your program results. When a nonfatal error occurs:

1. VS Pascal produces a message describing the error.
2. VS Pascal generates a trace-back report.
3. If you compiled and link-edited the program with the DEBUG option and the program *was not* executed with the DEBUG run-time option, VS Pascal produces a symbolic dump of the variables in the procedure experiencing the error.
4. If you compiled and link-edited the program with the DEBUG option and the program *was* executed with the DEBUG run-time option, VS Pascal will invoke the Interactive Debugging Tool as if a breakpoint had been encountered.

VS Pascal allows a specific number of nonfatal errors to occur before it terminates the program. This number is set by the ERRCOUNT run-time option (see Chapter 14, "Run-Time Options" on page 167). The default is 20.

How You Handle Run-Time Errors with the ONERROR Procedure

VS Pascal provides a way for you to gain control when a run-time error occurs. When run-time errors occur, the ONERROR procedure is called to perform any necessary action before issuing an error message. The VS Pascal library provides a default ONERROR routine (which does nothing).

You may write your own version of ONERROR and declare it as an EXTERNAL procedure. This allows you to control how errors are handled. Figure 46 shows the IBM-supplied include file that contains information that will help you produce your own ONERROR procedure. The member name in the include file is ONERROR.

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

TYPE
  ERRORTYPE = 1 .. 999;                (*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                    *)
```

Figure 46 (Part 1 of 2). Contents of ONERROR Include File

```
  ERRORSET = SET OF ERRORACTIONS;

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

Figure 46 (Part 2 of 2). Contents of ONERROR Include File

Upon entry to ONERROR, the parameter FERROR contains the number of the error encountered. See "Run-Time Messages" on page 243 to determine the message number corresponding to a particular error.

Note: Each error intercepted by the VS Pascal run-time environment consists of a unique three-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" for "informational", "E" for "error", or "S" for "severe error".

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

FACTION is a set variable that determines what action is to be taken. Upon invocation of ONERROR, FACTION describes the default actions that takes place after ONERROR returns. Examine your ONERROR procedure and decide whether you want to handle the error or let the default action take place. (The default actions for all errors passed to ONERROR are listed in Appendix C, "Run-Time Error Default Actions" on page 267.)

You can modify the FACTION parameter to fit your needs. Each member of FACTION and its function is listed below.

XHALT Causes the program to terminate. If XDEBUG is also in FACTION and the Interactive Debugging Tool is active, the debugging tool is invoked instead of terminating. If specified, the actions for XPMSG, XUMSG, and XTRACE are completed before the program terminates.

XPMSG Causes VS Pascal to display its diagnostic message for the error. If XHALT is in FACTION, a severe error (S-level) message is issued. If XDECERR is in FACTION, an error (E-level) message is issued. Otherwise, an informational (I-level) message is issued.

XUMSG Causes VS Pascal to display the user's message. If you set the XUMSG member of FACTION, you must also set FRETMSG with the text of the message. If XUMSG and XPMSG are both in FACTION, the VS Pascal diagnostic message is displayed before the user message.

XTRACE Causes VS Pascal to display a trace-back.

XDEBUG Causes the Interactive Debugging Tool to be invoked if active. If specified, the actions for XPMSG, XUMSG, and XTRACE are completed before invoking the debugger.

XDECERR Causes the run-time error counter to be decremented. The run-time error counter is initially set with the value specified in the ERRCOUNT option (see "ERRCOUNT Option" on page 168). When the error counter becomes equal to zero, the program is terminated with a severe error.

Figure 47 shows an example of a user interception of run-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) AND (FPROCNAME = 'HASHFNC') THEN
      BEGIN
        FACTION := [XUMSG];
        FRETMSG := 'INPUT DATA CONTAINS GARBAGE';
      END;
END;
```

Figure 47. Example of an Error Handling Routine Using ONERROR

Note: If your ONERROR procedure modifies a variable that is outside the scope of the ONERROR procedure, and you specify the OPTIMIZE compile-time option, the modification might not appear to have been made. This is because the variable might be in a register, and your modification effects only the copy of the variable in storage.

In the following program, if file F does not exist and the OPTIMIZE option is in effect, the program will print out "File opened OK" even though the file did not open successfully. This is because the variable FILE_ERROR was kept in a register, and though the storage copy was updated in ONERROR, the IF test used the value in the register.

```
PROGRAM ERROR;
VAR
  FILE_ERROR : INTEGER;
  F : TEXT;
%INCLUDE ONERROR;
PROCEDURE ONERROR;
BEGIN
  FACTION := [];
  FILE_ERROR := FERROR;
END;
BEGIN
  FILE_ERROR := 0;
  RESET(F);
  IF FILE_ERROR <> 0 THEN WRITELN('File error ',FILE_ERROR:1)
  ELSE WRITELN('File opened OK');
END.
```

Symbolic Variable Dump

When a program error or checking error occurs, VS Pascal can produce a symbolic dump of all variables local to the routine in which the error occurred. VS Pascal produces this dump when both:

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

In VM/CMS or MVS/TSO, the run-time library sends the variable dump to your terminal (unless you select another ddname with the ERRFILE run-time option).

In MVS batch, the run-time library sends the variable dump to SYSPRINT (unless you specify another ddname with the ERRFILE run-time option).

Chapter 7. How to Debug Your Program

This chapter describes how to use the Interactive Debugging Tool to find problems in your program.

Using the Interactive Debugging Tool

You can use the Interactive Debugging Tool in interactive mode and in batch mode. See Chapter 15, "Interactive Debugging Tool Commands" on page 173 for the debugging tool commands.

The Interactive Debugging Tool allows you to quickly debug VS Pascal programs without having to write debugging statements directly into your source code. Basic functions include:

- Tracing program execution
- Viewing the run-time values of program variables
- Breaking at intermediate points of execution
- Displaying statement frequency counting information.

You use VS Pascal source names to reference statements and data.

Using the Debugging Tool under VM

To use the debugging tool, you must follow these three steps:

1. Compile the unit you want to debug with the DEBUG option. Units you have compiled with the DEBUG option can be link-edited with units you have not compiled with the DEBUG option.
2. Link-edit your program, specifying the debugging library before the run-time library. If you use the PASCMOD EXEC (CMS), the debugging library is included by specifying the DEBUG option. (See "Step 2: How to Build a Load Module (Link-Editing)" on page 7.)
3. Execute the load module and specify DEBUG as a run-time option. DEBUG has two suboptions:

   - DEBUG(PROMPT), or simply DEBUG (the default), activates the Interactive Debugging Tool and prompts you to enter a debugging command. VS Pascal invokes the program only after you enter a debugging command that resumes execution (WALK or GO).
   - DEBUG(NOPROMPT) invokes the program immediately. The Interactive Debugging Tool is still active, but it prompts you to enter a debugging command only when a run-time error occurs.

   Run-time options must be terminated with a slash ("/"). (See Chapter 14, "Run-Time Options" on page 167.)

In the debugging environment, you can issue debugging commands directly from your terminal to examine variables in modules compiled with the DEBUG option. The output from your commands is sent to your terminal.

Using the Debugging Tool under MVS/TSO

To use the debugging tool, you must follow these three steps:

1. Compile the unit you want to debug with the DEBUG option. Units you have compiled with the DEBUG option can be link-edited with units that you have not compiled with the DEBUG option.
2. Link-edit your program, specifying the debugging library before the run-time library. If you use the PASCMOD CLIST (TSO), the debugging library is included by specifying the DEBUG option. (See "Step 2: How to Build a Load Module (Link-Editing)" on page 16.)
3. Execute the load module and specify DEBUG as a run-time option. DEBUG has two suboptions:
   - DEBUG(PROMPT), or simply DEBUG (the default), activates the Interactive Debugging Tool and prompts you to enter a debugging command. VS Pascal invokes the program only after you enter a debugging command that resumes execution (WALK or GO).
   - DEBUG(NOPROMPT) invokes the program immediately. The Interactive Debugging Tool is still active, but it prompts you to enter a debugging command only when a run-time error occurs.

   Run-time options must be terminated with a slash ("/"). (See Chapter 14, "Run-Time Options" on page 167.)

In the debugging environment, you can issue debugging commands directly from your terminal to examine variables in modules compiled with the DEBUG option.

Using the Debugging Tool under MVS Batch

Before debugging a program under MVS batch, update the appropriate cataloged procedure by:

1. Specifying DEBUG as a compile-time option
2. Concatenating the debug library before the run-time library
3. Specifying the DEBUG run-time option and defining the SYSIN data set in the GO step.

SYSIN is used to input the debugging commands, and must include all of the debugging commands you want executed during the debugging session. The output from these commands is sent to SYSPRINT.

Qualification

A qualification consists of a unit name and a routine name. The debugging tool uses the *current qualification* as the default to retrieve information for commands. The current qualification consists of the names of the unit and the routine that were active when the debugging tool gained control.

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

Listing the Debugging Commands

If you need a listing of the debugging commands, enter HELP or ?, and VS Pascal lists the debugging commands.

About Breakpoints

A breakpoint is a special command that forces a program to stop at a given location. You can then examine the values of variables and the contents of storage. In this way, you can verify the correct operation of the program at selected locations. You can also execute debugging commands when a breakpoint is encountered.

About Statement Counting

Before you can view statement counting information, you must use either the SET COUNT ON debugging command or the COUNT run-time option. To view counting information for a routine, counting must be set on *before* entry into the routine.

Sample Debugging Terminal Session

Figure 48 shows a sample program used in the following debugging examples.

```
   :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*)
   :   SAVEINDEX  : PRIMERANGE;            (*Used to remember last used  *)
   :                                       (*spot in Prime               *)
   :   TESTNUMBER : INTEGER;               (*Test value for primeness    *)
   :   PRIMEINDEX : PRIMERANGE;            (*Index into prime table      *)
   :FUNCTION ISPRIME(TESTVAL : INTEGER) : BOOLEAN;
   :VAR
   :   QUOTIENT,                           (*TestVal div Prime           *)
   :   REMAINDER : INTEGER;                (*Test value for primeness    *)
   :   PRIMEINDEX : PRIMERANGE;            (*Index into prime table      *)

   :BEGIN                                  (*function 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 ot TestVal and a known prime      *)
 3 :      QUOTIENT := TESTVAL DIV PRIME[PRIMEINDEX];
 4 :      REMAINDER := TESTVAL - QUOTIENT * PRIME [PRIMEINDEX];
 5 :   UNTIL (REMAINDER=0) OR (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, else it       *)
 8 :     ISPRIME := TRUE;                  (*is prime                    *)
   :END;
   :
   :BEGIN                                  (*program PrimGen             *)

 I :   PRIME [1]  := 2;                    (*First prime is two          *)
 2 :   PRIME [2]  := 3;                    (*Second prime is three       *)
 3 :   PRIME [3]  := 5;                    (*Third prime is five         *)
 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 candidate is a prime then*)
   :         BEGIN                         (*Save it in the next place   *)
 8 :            SAVEINDEX := SUCC(SAVEINDEX);   (*in the Prime table     *)
 9 :            PRIME[SAVEINDEX] := TESTNUMBER; (*                       *)
   :         END;
I0 :    UNTIL SAVEINDEX = HIGHEST(PRIMERANGE);
   :

   :   (*Print list of primes, 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 *)
14 :           WRITELN;                      (*then skip to next line   *)
   :       END;
   :END.                                   (*program PrimGen            *)
```

Figure 48. Sample Program for a Debugging Session

Figure 49 shows how to compile, link-edit, and run the sample program. The commands you enter are preceded with ===>.

```
===> VSPASCAL PRIMGEN ( DEBUG
 Invoking VS Pascal Release 2.0
  Starting Language Analysis Pass...
  Starting Optimization Pass...
  Starting Code Generation Pass...
  No Compiler Detected Errors

 Source lines:    57;  Total time:   0.18 seconds;  Total rate:  19000 LPM
Ready;
===> PASCMOD PRIMGEN ( DEBUG
Ready;
===> FILEDEF OUTPUT TERMINAL
Ready;
===> PRIMGEN DEBUG COUNT/
```

Figure 49. Compiling, Link-Editing, and Executing a Program with the DEBUG Option

Figure 50 shows how to issue the HELP command of the debugging tool. The command you enter is preceded with ===>.

```
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> ?
?
 Name (minimum abbreviation is in capital letters)
  ?               This command list
  , variable      Display a variable
  , hexconstant   Display memory
  Break           Set a breakpoint
  CLEAR           Remove all breakpoints
  Cms             Enter CMS subset mode
  Display         Display current resume point
  Display Breaks  Display currently set breakpoints
  Display Counts  Display current execution counts
  Display Equates Display currently set equates
  END             Halt your program

  Equate          Set an identifier to a literal value
  Go              Continue executing your program
  Help            This command list
  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
```

Figure 50. The HELP Command of the Interactive Debugging Tool

Figure 51 shows how to set and display breakpoints. The commands you enter are preceded with ===>.

```
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> BREAK 8
BREAK 8
 PRIMGEN/<MAIN-PROGRAM>/8
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> BREAK 12 EVERY 10 FROM 20 TO 50 (D B; GO)
BREAK 12 EVERY 10 FROM 20 TO 50 (D B; GO)
 PRIMGEN/<MAIN-PROGRAM>/12
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> BREAK END
BREAK END
 PRIMGEN/<MAIN-PROGRAM>/END
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> DISPLAY BREAKS
DISPLAY BREAKS
 Breakpoint 1 is set at
 Module    => PRIMGEN
 Procedure => <MAIN-PROGRAM>
 Statement =>    8

 Breakpoint 2 is set at
 Module    => PRIMGEN
 Procedure => <MAIN-PROGRAM>
 Statement =>   12
 Every     =>   10
 From      =>   20
 To        =>   50
 Count     =>    0
 Command   =>  D B; GO

 Breakpoint 3 is set at
 Module    => PRIMGEN
 Procedure => <MAIN-PROGRAM>
 Statement =>  END
```

Figure 51. Setting and Displaying Breakpoints

Figure 52 shows how to enter CMS mode during a debugging session. The command you enter is preceded with ===>.

```
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> CMS
CMS subset
Ready;
===> TIME
FRI 08/05/88 16:28:09 DAY 218 88218 080588 162809 880805 1628  4:28-PM
Ready;
===> RETURN
```

Figure 52. Entering CMS Mode

Figure 53 shows how to issue the GO and WALK commands for statement walking. The commands you enter are preceded with ===>.

```
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> GO
GO
 Stopped at PRIMGEN/<MAIN-PROGRAM>/8
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> WALK
WALK
 Stopped at PRIMGEN/<MAIN-PROGRAM>/9
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> WALK
WALK
 Stopped at PRIMGEN/<MAIN-PROGRAM>/10
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> GO
GO
 Stopped at PRIMGEN/<MAIN-PROGRAM>/8
```

Figure 53. The GO and WALK Debugging Commands

Figure 54 shows how to issue the LISTVARS command to list all the variables. The commands you enter are preceded with ===>.

```
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> SET ATTR ON
SET ATTR ON
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> LISTVARS
LISTVARS
 Variables for procedure: <MAIN-PROGRAM>
 Data Type    : ARRAY
 Length       :          400
 Memory Class : LOCAL AUTOMATIC
 Declared In  : <MAIN-PROGRAM>
  PRIME (00060930)
 000000 00000002 00000003 00000005 00000007 '.................'
 000010 FEFEFEFE FEFEFEFE FEFEFEFE FEFEFEFE '.................'
 (00000020 through 0000018F same as above)
 Data Type    : INTEGER
 Memory Class : LOCAL AUTOMATIC
 Declared In  : <MAIN-PROGRAM>
  PRIMEINDEX =  uninitialized
 Data Type    : INTEGER
 Memory Class : LOCAL AUTOMATIC
 Declared In  : <MAIN-PROGRAM>
  SAVEINDEX = 4
 Data Type    : INTEGER
 Memory Class : LOCAL AUTOMATIC
 Declared In  : <MAIN-PROGRAM>
  TESTNUMBER = 11
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> SET ATTR OFF
SET ATTR OFF
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> LISTVARS
```

Figure 54 (Part 1 of 2). Listing Variables in a Debugging Session

```
LISTVARS
 Variables for procedure: <MAIN-PROGRAM>
  PRIME (00060930)
 000000 00000002 00000003 00000005 00000007 '.............'
 000010 FEFEFEFE FEFEFEFE FEFEFEFE FEFEFEFE '.............'
 (00000020 through 0000018F same as above)
  PRIMEINDEX =  uninitialized
  SAVEINDEX = 4
  TESTNUMBER = 11
 Debug(PRIMGEN/<MAIN-PROGRAM>):
,PRIME(.4.)
  PRIME(.4.) = 7
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> ,TESTNUMBER (ATTR
,TESTNUMBER (ATTR
 Data Type    : INTEGER
 Memory Class : LOCAL AUTOMATIC
 Declared In  : <MAIN-PROGRAM>
  TESTNUMBER  = 11
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> ,'60930'X
,'60930'X
 (00060930)
 000000 00000002 00000003 00000005 00000007 '.............'
```

Figure 54 (Part 2 of 2). Listing Variables in a Debugging Session

Figure 55 shows how to execute with the trace mode of the debugging tool set to ON. The commands you enter are preceded with ===>.

```
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> SET TRACE ON
SET TRACE ON
 Program trace is currently on -- output is to the terminal
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> GO
GO
 Resuming PRIMGEN/<MAIN-PROGRAM>
 ======>  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>):
```

Figure 55 (Part 1 of 2). Executing with Trace Mode On

```
===> WALK
WALK
 Stopped at PRIMGEN/<MAIN-PROGRAM>/9
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> WALK
WALK
 ======>  10
 Stopped at PRIMGEN/<MAIN-PROGRAM>/10
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> WALK
WALK
 ======>   6-7
 Stopped at PRIMGEN/<MAIN-PROGRAM>/6
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> SET TRACE OFF
SET TRACE OFF
 Program trace is currently off
```

Figure 55 (Part 2 of 2). Executing with Trace Mode On

Figure 56 shows how to get a statement counting summary. The command you enter is preceded with ===>.

```
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> DISPLAY COUNTS
DISPLAY COUNTS
 VS PASCAL  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  :5          8-9  :3           10 :4
     11 :0         12-13 :0            14 :0

ISPRIME IN PRIMGEN CALLED 4 TIME(S)
 FROM-TO:COUNT    FROM-TO:COUNT    FROM-TO:COUNT    FROM-TO:COUNT
     1  :4          2-5  :5            6  :4            7  :1
     8  :3
```

Figure 56. Displaying a Statement Counting Summary

Figure 57 shows how to get a trace-back and a sample trace-back report. (See "Reading a VS Pascal Trace-Back Report" on page 73 for more information.) The commands you enter are preceded with ===>.

```
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> GO
GO
 Stopped at PRIMGEN/<MAIN-PROGRAM>/8
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> BREAK ISPRIME/END
BREAK ISPRIME/END
 PRIMGEN/ISPRIME/END
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> GO
GO
 Stopped at PRIMGEN/ISPRIME/END
 Debug(PRIMGEN/ISPRIME):
===> TRACE
TRACE
      TRACE BACK OF CALLED ROUTINES
 ROUTINE                          STMT AT ADDRESS IN MODULE
 ISPRIME                             8    00020190   PRIMGEN
 <MAIN-PROGRAM>                      7    000203F2   PRIMGEN
 VSPASCAL                                 00020922

 Debug(PRIMGEN/ISPRIME):
===> RESET ISPRIME/END
RESET ISPRIME/END
 Breakpoint at PRIMGEN/ISPRIME/END has been removed
 Debug(PRIMGEN/ISPRIME):
===> GO
GO
 Stopped at PRIMGEN/<MAIN-PROGRAM>/8
```

Figure 57. Using the TRACE Debugging Command to Get a Trace-Back Report

Figure 58 shows the commands to set and use equates. The commands you enter are preceded with ===>.

```
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> EQUATE TN ,TESTNUMBER
EQUATE TN ,TESTNUMBER
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> TN

TN
  TESTNUMBER = 19
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> EQUATE SI ,SAVEINDEX
EQUATE SI ,SAVEINDEX
 Debug(PRIMGEN/<MAIN-PROGRAM>):

===> SI
SI
  SAVEINDEX = 7
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> DISPLAY EQUATES
DISPLAY EQUATES
 TN ==> ,TESTNUMBER
 SI ==> ,SAVEINDEX
```

Figure 58. Using Equates in a Debugging Session

Figure 59 shows the commands to remove breakpoints. The commands you enter are preceded with ===>.

```
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> DISPLAY BREAKS
DISPLAY BREAKS
 Breakpoint 1 is set at
 Module    => PRIMGEN
 Procedure => <MAIN-PROGRAM>
 Statement =>    8


 Breakpoint 2 is set at
 Module    => PRIMGEN
 Procedure => <MAIN-PROGRAM>
 Statement =>   12
 Every     =>   10
 From      =>   20
 To        =>   50
 Count     =>    0
 Command   =>  D B; GO


 Breakpoint 3 is set at
 Module    => PRIMGEN
 Procedure => <MAIN-PROGRAM>
 Statement =>  END

 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> GO
GO
 Stopped at PRIMGEN/<MAIN-PROGRAM>/8
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> RESET 8
RESET 8
 Breakpoint at PRIMGEN/<MAIN-PROGRAM>/8 has been removed
 Debug(PRIMGEN/<MAIN-PROGRAM>):
```

Figure 59 (Part 1 of 2). Removing Breakpoints

```
===> D B
D B
 Breakpoint 2 is set at
 Module    => PRIMGEN
 Procedure => <MAIN-PROGRAM>
 Statement =>   12
 Every     =>   10
 From      =>   20
 To        =>   50
 Count     =>    D
 Command   =>  D B; GO

 Breakpoint 3 is set at
 Module    => PRIMGEN
 Procedure => <MAIN-PROGRAM>
 Statement =>  END
```

Figure 59 (Part 2 of 2). Removing Breakpoints

Figure 60 shows the effects of the options on the BREAK command.

```
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> GO
GO
      2      3      5      7     11     13     17     19     23     29
 Stopped at PRIMGEN/<MAIN-PROGRAM>/12
===> D B
D B
 Breakpoint 2 is set at
 Module    => PRIMGEN
 Procedure => <MAIN-PROGRAM>
 Statement =>   12
 Every     =>   10
 From      =>   20
 To        =>   50
 Count     =>   20
 Command   =>  D B; GO

 Breakpoint 3 is set at
 Module    => PRIMGEN
 Procedure => <MAIN-PROGRAM>
 Statement =>  END

===> GO
GO
     31     37     41     43     47     53     59     61     67     71
 Stopped at PRIMGEN/<MAIN-PROGRAM>/12
```

Figure 60 (Part 1 of 3). Effects of BREAK Command Options

```
===> D B
D B
 Breakpoint 2 is set at
 Module    => PRIMGEN
 Procedure => <MAIN-PROGRAM>
 Statement =>   12
 Every     =>   10
 From      =>   20
 To        =>   50
 Count     =>   30
 Command   =>  D B; GO

 Breakpoint 3 is set at
 Module    => PRIMGEN
 Procedure => <MAIN-PROGRAM>
 Statement =>  END

===> GO
GO
     73     79     83     89     97    101    103    107    109    113
 Stopped at PRIMGEN/<MAIN-PROGRAM>/12

===> D B
D B
 Breakpoint 2 is set at
 Module    => PRIMGEN
 Procedure => <MAIN-PROGRAM>
 Statement =>   12
 Every     =>   10
 From      =>   20
 To        =>   50
 Count     =>   40
 Command   =>  D B; GO

 Breakpoint 3 is set at
 Module    => PRIMGEN
 Procedure => <MAIN-PROGRAM>
 Statement =>  END

===> GO
GO
    127    131    137    139    149    151    157    163    167    173
 Stopped at PRIMGEN/<MAIN-PROGRAM>/12
```

Figure 60 (Part 2 of 3). Effects of BREAK Command Options

```
===> D B
D B
 Breakpoint 2 is set at
 Module    => PRIMGEN
 Procedure => <MAIN-PROGRAM>
 Statement =>   12
 Every     =>   10
 From      =>   20
 To        =>   50
 Count     =>   50
 Command   =>  D B; GO

 Breakpoint 3 is set at
 Module    => PRIMGEN
 Procedure => <MAIN-PROGRAM>
 Statement =>  END

===> GO
GO
    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
 Stopped at PRIMGEN/<MAIN-PROGRAM>/END
 Debug(PRIMGEN/<MAIN-PROGRAM>):
===> CLEAR
CLEAR
 All breakpoints have been removed
```

Figure 60 (Part 3 of 3). Effects of BREAK Command Options

Figure 61 shows the statement counting summary.

```
 VS PASCAL  STATEMENT COUNTING SUMMARY                        PAGE1

<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
Ready;
```

Figure 61. Statement Counting Summary

Chapter 8. How to Use Interlanguage Communication

It is sometimes desirable to invoke subprograms (procedures) written in other programming languages: this is useful to obtain services not available directly in VS Pascal. It is also desirable to have a VS Pascal procedure called from a non-VS Pascal program: this allows you to take advantage of VS 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.

If you are running Pascal under the XA environment in 31-bit addressing mode, you must be sure that any language that calls VS Pascal or any language that VS Pascal calls resides in the same address space as the VS Pascal code (either above or below the 16-megabyte address line). For example, if a FORTRAN program is below the 16-megabyte address line and calls a VS Pascal routine, the VS Pascal routine must also be below the 16-megabyte address line. If a VS Pascal program resides above the 16-megabyte address line and calls a COBOL program, the COBOL program must also be above the 16-megabyte address line.

We can divide interlanguage communication into two classes:

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

Your options are summarized in Figure 62.

Language	VS Pascal as the Calling Language	VS Pascal as the Called Language
Assembler	Define procedures and functions in VS Pascal using the FORTRAN or the EXTERNAL directive. If you use EXTERNAL, you will be able to specify an arbitrary VS Pascal parameter list.	Use a V-type constant in the assembler routine to define the VS Pascal entry point. You must declare the VS Pascal procedure with the EXTERNAL, MAIN, or REENTRANT directive. After the last call to a MAIN or REENTRANT procedure you must call PSCLHX to clean up the VS Pascal run-time environment.
FORTRAN	Define procedures and functions in VS Pascal using the FORTRAN directive. This enables you to call a subprogram written in FORTRAN. In general, you must initialize the FORTRAN run-time environment. See Figure 71 on page 109 for an example, and the appropriate FORTRAN manual (see "FORTRAN" on page 296) for further information.	Use a call statement in FORTRAN to call the VS Pascal procedure. The VS Pascal procedure must be declared with the MAIN or REENTRANT directive. After the last call to a VS Pascal procedure, you must call PSCLHX to clean up the VS Pascal run-time environment.

Figure 62 (Part 1 of 2). VS Pascal Communication with Other Languages

Language	VS Pascal as the Calling Language	VS Pascal as the Called Language
COBOL	Define procedures and functions in VS Pascal using the FORTRAN directive. This enables you to call a subprogram written in COBOL. You may want to call ILBOSTP0 before calling a COBOL program. See the appropriate COBOL manual (see "COBOL" on page 295) for details.	Use a call statement in COBOL to call the VS Pascal procedure. COBOL should be compiled with the NODYNAM option and the call must be a call of a literal. The VS Pascal procedure must be declared with the MAIN or REENTRANT directive. After the last call to a VS Pascal procedure, you must call PSCLHX to clean up the VS Pascal run-time environment.
PL/I	Define procedures and functions in VS 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. See the appropriate PL/I manual (see "PL/I" on page 296) for details.	Use a call statement in PL/I to call a VS Pascal procedure. You must declare the VS Pascal procedure with the MAIN or REENTRANT directive. The PL/I procedure should specify the VS Pascal procedure as an EXTERNAL. After the last call to a VS Pascal procedure, you must call PSCLHX to clean up the VS Pascal run-time environment.

Figure 62 (Part 2 of 2). VS Pascal Communication with Other Languages

When to Use the EXTERNAL Directive

The EXTERNAL directive is used to call a routine outside the lexical scope of the caller (such as another unit). Both routines must use Pascal linkage conventions (for example, when a Pascal program calls an assembler routine which then calls a Pascal routine using the general interface). Both of the called routines must have been defined as EXTERNAL.

While many units may call an EXTERNAL routine, its body will be contained only in one unit. The formal parameters defined in the EXTERNAL routine must match those in the unit where the routine is defined.

For more information see the *VS Pascal Language Reference*.

When to Use the MAIN Directive

The MAIN directive can be used on routines called by PL/I, FORTRAN, and COBOL, and by assembler routines which do not need the general interface. The MAIN directive is used to identify a Pascal procedure that can be invoked as if it were a main program.

The first call to a MAIN directive causes a VS Pascal environment to be created. It is your responsibility to remove this environment, and the procedure PSCLHX has been provided for this purpose (see "When to Use the PSCLHX Procedure" on page 99 for more information about the PSCLHX procedure).

When to Use the REENTRANT Directive

The REENTRANT directive can be used on routines called from PL/I, FORTRAN, and COBOL, and by assembler routines which do not need the general interface. The REENTRANT directive is used to identify a VS Pascal procedure that may be invoked as if it were a main program, like a MAIN procedure. In addition, invocations of this procedure may be reentrant.

In order for this procedure to be reentrant, the first parameter of a procedure defined with REENTRANT must be an INTEGER passed by VAR. Before the first call to VS Pascal from a non-Pascal program, this variable must be set to zero (0). On subsequent calls the same variable must be passed back unaltered. You need not call the same procedure each time; you may call different procedures (just continue to pass this variable on each call).

The first call to a REENTRANT directive causes a VS Pascal environment to be created. It is your responsibility to remove this environment, and the procedure PSCLHX has been provided for this purpose, (see "When to Use the PSCLHX Procedure" for more information about the PSCLHX procedure).

For more information see the *VS Pascal Language Reference*.

When to Use the PSCLHX Procedure

The PSCLHX procedure is used to clean up the environment created by a call to a MAIN or REENTRANT directive. PSCLHX will free memory and close files used by the VS Pascal environment.

When called after invoking a MAIN procedure, the call to PSCLHX must pass zero (0) as an argument in the form:

```
PSCLHX(0)
```

When called after invoking a REENTRANT procedure, the call to PSCLHX must pass the same parameter passed to the initial call to the REENTRANT procedure. The call to PSCLHX is made in the form:

```
PSCLHX(x)
```

where *x* is the same parameter passed on the initial call to the REENTRANT procedure.

For more information see the *VS Pascal Language Reference*.

VS Pascal and Assembler

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

- *VS Pascal parameter passing conventions*: As described in Chapter 17, "VS Pascal Parameter Passing" on page 195, VS Pascal parameters are passed in a variety of ways, depending on their attributes.
- *The VS Pascal environment*: This is an arrangement of registers and control blocks used by VS Pascal to handle storage management and run-time error recovery. (See "Register Usage" on page 185.)

VS Pascal as the Caller to Assembler

Assembler routines can be called in two ways depending on your needs. The minimum interface (see "Writing an Assembler Routine with a Minimum Interface" on page 100) is simpler and faster, but may not provide enough function. The general interface (see "Writing an Assembler Routine with a General Interface" on page 101) is more flexible, but it imposes structuring requirements on your program.

Writing an Assembler Routine with a Minimum Interface

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

- It may not call a VS Pascal routine.
- It must be nonrecursive.
- If a program error should occur (such as divide by zero), the VS Pascal run-time environment will not recover properly and the results will be unpredictable.

When a VS Pascal 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 registers 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 back chain word in a save area defined by the assembler routine itself. Before returning to the VS Pascal routine, the registers must be restored to the values they contained when the assembler routine was invoked.

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

Note: The example below is not reentrant.

```
ANYNAME  CSECT
         ENTRY PROCNAME        declare routine name as an entry point
PROCNAME DS    0H              entry point to routine
         STM   14,12,12(13)    save VS Pascal registers in VS Pascal save area
         BALR  BASEREG,0       establish base register
         USING *,BASEREG
         ST    13,SAVEAREA+4   store VS Pascal 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 VS Pascal save area
         LM    14,12,12(13)    restore VS Pascal registers
         BR    14              return to VS Pascal
SAVEAREA DC    20F'0'          local save area
         END
```

Figure 63. Minimum Interface to an Assembler Routine (Skeletal Code to be Invoked from VS Pascal)

Figure 64 shows an example of VS Pascal as the caller to an assembler routine using the minimum interface.

VS Pascal program which invokes an Assembler routine named SUM:

```
PROGRAM FROMPSCL;                          (*VS 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 VS Pascal program:

```
SUM      CSECT
         USING *,15              establish addressability
         STM   14,12,12(13)      save caller's registers
         ST    13,SAVEAREA+4     save address of caller's 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     get address of caller's save area
         LM    14,12,12(13)      restore caller's registers
         BR    14                return
SAVEAREA DS    18F
         END
```

Figure 64. Example of VS Pascal as the Caller to an Assembler Routine Using the Minimum Interface

Writing an Assembler Routine with a General Interface

If an assembler routine has at least one of the following characteristics:

- It calls a VS Pascal routine
- It is recursive
- Program errors must be intercepted and diagnosed by the VS Pascal run-time environment

the general interface must be used.

Two assembler macros are available which are used to generate the prolog and epilog of an assembler routine with a general VS Pascal interface. The macro names are PROLOG and EPILOG and their forms are described in the figures below.

The PROLOG Macro:

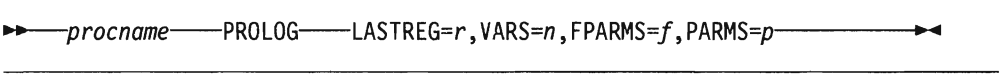


Figure 65. PROLOG Macro Syntax Diagram

Where	Represents
procname	The entry point name of the routine.
PROLOG	Keyword.
LASTREG=*r*	The highest register to be modified by the routine; *r* is a number between 3 and 12, inclusive. *Default*: 12.
VARS=*n*	The number of bytes (*n*) required for any local data, including passed-in parameters. *Default*: 0.
FPARMS=*f*	The number of bytes (*f*) for any passed-in parameters. *Default*: The value (*n*) specified for VARS.
PARMS=*p*	The number of bytes (*p*) required for the largest parameter list to be built within the routine. *Default*: 0.

The PROLOG macro preserves any registers that are to be modified and allocates storage for the dynamic storage area (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.

After entering a routine, but before executing the PROLOG code, the following registers are expected to contain the indicated data:

Register	*Contents*
1	Address of the parameter list built by the caller, which is 144 bytes into the dynamic storage area (DSA) to be used by the called routine.
11	Address of the dynamic storage area (DSA) of the main program.
12	Address of the VS Pascal communication work area (PCWA).
13	Address of the dynamic storage area (DSA) of the calling routine.
14	Return address.
15	Address of the start of the called routine.

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

Register	*Contents*
0	Unchanged.
1	Address of an area of storage in which parameter lists can be built to pass to other routines.
2	Base register for the first 4096 bytes of code within the invoked routine.
3–11	Unchanged.
12	Unchanged.
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.
14	Unchanged.
15	Unpredictable.

Note: Passed-in parameters start at offset 144 from register 13 after the prolog has been executed.

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 before returning from the routine.

The EPILOG Macro:

Figure 66. EPILOG Macro Syntax Diagram

Where	**Represents**
EPILOG	Keyword.
DROP=YES|NO	Whether or not register 2 is to be dropped as a base register after the epilog is executed. *Default*: YES.

The EPILOG macro restores the saved registers, then branches back to the calling routine. In order for the epilog to execute properly, register 13 must have the same contents as was established by the prolog. The macro will cause register 2 to be dropped as a base register unless DROP=NO is specified.

A skeleton of a general-interface assembler language routine, which may be called by a VS Pascal program, is shown in Figure 67.

```
* 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.
* 'lastreg' is the highest register (up to 12) which will be modified
* 'varsize' is the storage required for the local variables
* 'parmsize' is the length of the passed-in parameters
* '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,FPARMS=parmsize,PARMS=plist
         .
         .                        <== insert code here
         .
*
         EPILOG
         END
```

Figure 67. General Interface to an Assembler Routine

Receiving Parameters from Routines

Parameters received from a VS Pascal routine are mapped within a list in the manner described in Chapter 17, "VS Pascal Parameter Passing" on page 195. At invocation, register 1 contains the address of this list.

Assembler as the Caller to VS Pascal

An assembler language routine that was invoked from a VS Pascal program may call a VS Pascal procedure without setting up the VS Pascal run-time environment provided that:

- The assembler routine was called from a VS Pascal routine.
- The VS Pascal routine to be called is declared using the EXTERNAL directive.

See Figure 68 on page 105 as an example.

Before making the call to a VS Pascal procedure from an assembler routine, 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 Chapter 17, "VS Pascal Parameter Passing" on page 195.

If the assembler routine was not invoked from a VS Pascal routine, then the VS Pascal run-time environment must be set up before entering the VS Pascal routine. To do this, the VS Pascal procedure must be declared using the MAIN or REENTRANT directive. (See "When to Use the REENTRANT Directive" on page 98 and "When to Use the MAIN Directive" on page 98.)

At the point of call, register 12 must contain the address of the VS Pascal communication work area (PCWA). This will be the case if the assembler routine was invoked from a VS Pascal 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.

Sample Assembler Routine

Figure 68 shows the VS Pascal description of an assembler routine which may be called from a VS Pascal program, and the described assembler routine. The assembler routine executes an MVS TPUT macro to write a line of text to your terminal.

VS Pascal program that calls an Assembler routine

```
PROGRAM PASCAL;
  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);
    EXTERNAL;
  PROCEDURE ERROR;
  BEGIN
    WRITELN(OUTPUT, MESSAGE, ', RETURN CODE = ', RETCODE)
  END;
BEGIN
 TPUT('Test assembler call',19);
END.
```

Assembler routine being invoked from a VS Pascal program

```
  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
```

Figure 68 (Part 1 of 2). Sample Assembler Routine

```
         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 68 (Part 2 of 2). Sample Assembler Routine

Calling a VS Pascal Main Program from an Assembler Routine

A VS Pascal 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.

Figure 69 shows an example of an assembler routine as the caller to a VS Pascal procedure.

The VS Pascal procedure to be called:

```
SEGMENT SQUARE;
PROCEDURE SQUARE(VAR X : REAL);
  MAIN;
PROCEDURE SQUARE;
BEGIN
  X := X * X
END; .
```

Assembler routine to call VS Pascal procedure:

```
TOSQ     CSECT
         USING *,15              establish addressability
         STM   14,12,12(13)      save caller's registers
         ST    13,SAVEAREA+4     save address of caller's 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 VS Pascal procedure
         BALR  14,15             call SQUARE
```

Figure 69 (Part 1 of 2). Example of Assembler as the Caller to a VS Pascal Procedure

```
         LA    1,PLIST2          reg 1 points to parameter list
         L     15,=V(PSCLHX)     load address of VS Pascal procedure
         BALR  14,15             call PSCLHX to terminate environment
         L     13,SAVEAREA+4     get address of caller's save area
         LM    14,12,12(13)      restore caller's registers
         BR    14                return
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 69 (Part 2 of 2). Example of Assembler as the Caller to a VS Pascal Procedure

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

Calling a VS Pascal Program from an Assembler Main Procedure

Figure 70 shows an example of calling a VS Pascal program from an assembler routine.

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
```

Figure 70 (Part 1 of 2). Example of Calling a VS Pascal Program from an Assembler Routine

```
        .
        .
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 MVS (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 70 (Part 2 of 2). Example of Calling a VS Pascal Program from an Assembler Routine

VS Pascal and FORTRAN

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

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

VS Pascal as the Caller to FORTRAN

Figure 71 shows an example of VS Pascal as the caller to a FORTRAN routine.

VS Pascal program that calls a FORTRAN subroutine:

```
PROGRAM FROMPSCL;                        (*VS Pascal program heading   *)
  (*THE FOLLOWING ROUTINE IS IN ASSEMBLER LANGUAGE*)
  PROCEDURE INITFORT;
      FORTRAN;
  (*THE FOLLOWING ROUTINE IS IN FORTRAN*)
  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      *)
  INITFORT;                              (*Initialize VS FORTRAN        *)
  FOR J := 1 TO 10 DO                    (*Loop through ten values      *)
    SUM(I,J);                            (*Compute the next sum         *)
END .                                    (*FROMPSCL                     *)
```

ASSEMBLER routine to initialize the FORTRAN environment (this routine may be dependent on the FORTRAN release, and is not reentrant).

```
  LANGCALL CSECT
           ENTRY INITFORT
  INITFORT DS    0H                    INITIALIZE FORTRAN ROUTINE
           STM   14,12,12(13)          SAVE CALLER'S ENVIRONMENT AND
           BALR  2,0                     ESTABLISH ADDRESSABILITY
           USING *,2                     REGISTER 2 IS THE BASE
           ST    13,SAVEAREA+4           REGISTER
           LA    13,SAVEAREA
           LA    1,PARMLIST            POINT TO PARAMETERS
           L     15,=V(FEIN#)          INITIALIZE FORTRAN
           BALR  14,15                   EXECUTION-TIME ENVIRONMENT
```

Figure 71 (Part 1 of 2). Example of VS Pascal as the Caller to a FORTRAN Routine

```
          L     13,4(13)                RESTORE CALLER'S ENVIRONMENT
          L     14,12,12(13)
          BR    14                      RETURN TO CALLER
          LTORG
SAVEAREA  DC    20F'0'                  REGISTER SAVE AREA
PARMLIST  DC    A(PARMAREA+X'80000000')
PARMAREA  DC    Y(L'PARMSTRG)
PARMSTRG  DC    C'OPTION,...'           EXECUTION-TIME OPTIONS LIST
          END
```

FORTRAN subroutine:

```
      SUBROUTINE SUM(I,J)
      I = I + J
      WRITE (6,4) I
4     FORMAT (' The current running sum is ',I3)
      RETURN
      END
```

Figure 71 (Part 2 of 2). Example of VS Pascal as the Caller to a FORTRAN Routine

The FORTRAN directive instructs VS Pascal to use exactly the same calling conventions used by FORTRAN. This restricts the form of the parameter list, namely, you may not pass a parameter by value; you may only pass a parameter by VAR or by CONST. If you choose the latter mechanism, the FORTRAN subprogram must not modify the parameter.

With most calls to FORTRAN you must initialize the FORTRAN execution-time environment. You must write an assembler language routine to do this, because most FORTRAN entry points include a pound sign (#) which is not valid in a VS Pascal identifier. Also, the assembler routine can ensure that the desired execution-time options are in effect for the FORTRAN subroutines you want to use.

Run-time errors that occur during the execution of the FORTRAN program will be handled by the FORTRAN error-handling subroutines. See the appropriate FORTRAN publication (see "FORTRAN" on page 296) for details.

FORTRAN as the Caller to VS Pascal

VS Pascal permits a FORTRAN program to call a VS Pascal procedure as a subprogram. To do this, you specify the VS Pascal procedure with the MAIN directive (see "When to Use the MAIN Directive" on page 98).

Figure 72 shows an example of a FORTRAN routine as the caller to VS Pascal using the MAIN directive.

VS Pascal procedure to be called from FORTRAN using the MAIN directive:

```
SEGMENT SQUARE;
PROCEDURE SQUARE(VAR E : INTEGER; VAR X : REAL);
  MAIN;
PROCEDURE SQUARE;
BEGIN
  X := X * X
END;.
```

FORTRAN program that calls a VS Pascal procedure:

```
      REAL*8 AREAL
      INTEGER * 4 SAVE
      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 72. Example of FORTRAN as the Caller to a VS Pascal Procedure

VS Pascal also permits a FORTRAN program to call a VS Pascal procedure as a subprogram using the REENTRANT directive (see "When to Use the REENTRANT Directive" on page 98).

VS Pascal procedure to be called from FORTRAN program using the REENTRANT directive:

```
SEGMENT SQUARE;
PROCEDURE SQUARE(VAR E : INTEGER; VAR X : REAL);
  REENTRANT;
PROCEDURE SQUARE;
BEGIN
  X := X * X
END;.
```

Figure 73 (Part 1 of 2). Example of FORTRAN as the Caller to a VS Pascal Procedure

Reentrant FORTRAN program that calls a VS Pascal procedure:

```
      REAL*8 AREAL
      INTEGER*4 SAVE
      AREAL = 4.0
      SAVE = 0
      CALL SQUARE(SAVE,AREAL)
      PRINT 1, AREAL
      CALL SQUARE(SAVE,AREAL)
      PRINT 1, AREAL
      CALL SQUARE(SAVE,AREAL)
      PRINT 1, AREAL

      CALL SQUARE(SAVE,AREAL)
      PRINT 1, AREAL
C        TERMINATE PASCAL ENVIRONMENT
      CALL PSCLHX(SAVE)
      STOP
    1 FORMAT (F12.0)
      END
```

Figure 73 (Part 2 of 2). Example of FORTRAN as the Caller to a VS Pascal Procedure

It is your responsibility to clean up the VS Pascal environment; this is done by invoking PSCLHX (see "When to Use the PSCLHX Procedure" on page 99).

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

VS Pascal and COBOL

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

VS Pascal as the Caller to COBOL

Figure 74 shows an example of VS Pascal as the caller of a COBOL routine.

VS Pascal program that calls a COBOL subprogram:

```
PROGRAM FROMPSCL;                        (*VS 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 74. Example of VS Pascal as the Caller of a COBOL Routine

The FORTRAN directive instructs VS Pascal to use exactly the same calling conventions used 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 only pass a parameter by VAR or by CONST. If you choose the latter mechanism, the COBOL subprogram must not modify the parameter.

Run-time errors that occur during the execution of the COBOL program will be handled by the VS Pascal run-time support routines. VS 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 VS Pascal calling procedure as a main entry point of a COBOL program. See the appropriate COBOL publication (see "COBOL" on page 295).

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

COBOL as the Caller to VS Pascal

VS Pascal permits a COBOL program to call a VS Pascal procedure as a subprogram. To do this, you specify the VS Pascal procedure with the MAIN directive or the REENTRANT directive (see "When to Use the MAIN Directive" on page 98). Subsequent calls will use the same environment created by the first call.

Figure 75 shows an example of COBOL as the caller of VS Pascal using the MAIN directive.

VS Pascal procedure that is called from COBOL using the MAIN directive:

```
SEGMENT SQUARE;
PROCEDURE SQUARE(VAR X : REAL);
  MAIN;
PROCEDURE SQUARE;
BEGIN
  X := X * X
END; .
```

COBOL program that calls a VS Pascal procedure:

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. TOSQ.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER. IBM-370.
       OBJECT-COMPUTER. IBM-370.
       DATA DIVISION.
       WORKING-STORAGE 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 75. Example of COBOL as the Caller to a VS Pascal Procedure Using the MAIN Directive

VS Pascal also permits a COBOL program to call a VS Pascal procedure as a subprogram using the REENTRANT directive (see "When to Use the REENTRANT Directive" on page 98).

REENTRANT VS Pascal procedure that is called from COBOL:

```
SEGMENT SQUARE;
PROCEDURE SQUARE(VAR E : INTEGER; VAR X : REAL);
  REENTRANT;
PROCEDURE SQUARE;
BEGIN
  X := X * X
END; .
```

COBOL program that calls a VS Pascal procedure:

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

Figure 76. Example of COBOL as the Caller to a REENTRANT VS Pascal Procedure

It is your responsibility to clean up the VS Pascal environment; this is done by invoking PSCLHX (see "When to Use the PSCLHX Procedure" on page 99).

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

VS Pascal and PL/I

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

VS Pascal as the Caller to PL/I

Figure 77 shows an example of VS Pascal as the caller of PL/I.

VS Pascal program that calls a PL/I procedure:

```
PROGRAM FROMPSCL;                    (*VS 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:

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

Figure 77. Example of VS Pascal as the Caller of a PL/I Routine

The FORTRAN directive instructs VS Pascal to use exactly the same calling conventions used by FORTRAN. PL/I will employ FORTRAN calling conventions if "FORTRAN" is specified in the OPTIONS clause. See *OS PL/I Optimizing Compiler: Programmer's Guide* for details.

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

PL/I as the Caller to VS Pascal

VS Pascal permits a PL/I program to call a VS Pascal procedure as a subprogram. To do this, you specify the VS Pascal procedure with the MAIN directive (see "When to Use the MAIN Directive" on page 98).

Figure 78 shows an example of PL/I as the caller of VS Pascal.

VS Pascal procedure that is called from PL/I using the MAIN directive:

```
SEGMENT SQUARE;
PROCEDURE SQUARE(VAR X : REAL);
  MAIN;
PROCEDURE SQUARE;
BEGIN
  X := X * X
END; .
```

PL/I program that calls a VS Pascal procedure:

```
TOSQ: PROC OPTIONS(MAIN);
     DCL SQUARE ENTRY EXTERNAL OPTIONS(ASM);
     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 78. Example of PL/I as the Caller to a VS Pascal Procedure Using the REENTRANT Directive

The REENTRANT directive may be used in place of the MAIN directive if the program must be reentrant (see "When to Use the REENTRANT Directive" on page 98).

Figure 79 shows an example of PL/I as the caller of a VS Pascal procedure using the REENTRANT directive.

VS Pascal procedure that is called from PL/I using the REENTRANT directive:

```
SEGMENT SQUARE;
PROCEDURE SQUARE(VAR E : INTEGER; VAR X : REAL);
  REENTRANT;
PROCEDURE SQUARE;
BEGIN
  X := X * X
END; .
```

Reentrant PL/I program that invokes a VS 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 79. Example of PL/I as the Caller to a VS Pascal Procedure Using the REENTRANT Directive

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

To terminate the VS Pascal environment set up by the REENTRANT procedure, PSCLHX should be called (see "When to Use the PSCLHX Procedure" on page 99) with the variable that contains the address. See Figure 79 for an example.

Data Type Comparisons

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 VS Pascal; however, you can often create new user data types in VS 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 80 compares VS Pascal data types with the equivalent ones in FORTRAN, COBOL, and PL/I.

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

VS Pascal	FORTRAN	COBOL	PL/I
ARRAY	Dimensioned variable	OCCURS	Dimensioned variable
@ id	Not applicable	Not applicable	POINTER
BOOLEAN	LOGICAL*1	Not applicable	BIT(8) ALIGNED
CHAR	CHARACTER*1	PIC X	CHAR
GCHAR	Not applicable	PIC G	GRAPHIC(1)
GSTRING(n)	Not applicable	Not applicable	GRAPHIC(n) VARYING
INTEGER	INTEGER*4	PIC S999999999 USAGE IS COMPUTATIONAL	FIXED BINARY(31,0)
PACKED -32768..32767	INTEGER*2	PIC S9999 USAGE IS COMPUTATIONAL	FIXED BINARY(15,0)
PACKED 0..65536	Not applicable	Not applicable	Not applicable
PACKED -128..127	Not applicable	Not applicable	FIXED BINARY(7,0)
PACKED 0..255	Not applicable	Not applicable	Not applicable
PACKED ARRAY[1..n] OF CHAR	CHARACTER*n	PIC X(n) or PIC X OCCURS n TIMES	CHAR(n)
PACKED ARRAY[1..n] OF GCHAR	Not applicable	PIC G(n)	GRAPHIC(n)
REAL	REAL*8	COMPUTATIONAL-2	REAL FLOAT DEC(16)
RECORD	Not applicable	Record	Structure
SET OF 0..n	Not applicable	Not applicable	BIT(n+1)
SHORTREAL	REAL*4	COMPUTATIONAL-1	REAL FLOAT DEC(6)
STRING(n)	Not applicable	Not applicable	CHAR(n) VARYING
SPACE	Not applicable	Not applicable	AREA

Figure 80. Data Type Equivalences between Different Languages

Chapter 9. Interfacing With IMS

Two routine directives make it possible for VS Pascal to interface with IMS:

- VS Pascal routines declared with the GENERIC directive can call other routines written in IMS DL/I.
- VS Pascal routines declared with the REENTRANT directive can be called by IMS.

An IMS program that calls a VS Pascal procedure must be defined with LANG=PASCAL in the PSBGEN statement.

This chapter provides an example of how to use the GENERIC and REENTRANT directives to interface with IMS. For more information on GENERIC and REENTRANT, see *VS Pascal Language Reference*.

Note: VS Pascal programs that interface with IMS must execute in the 31-bit addressing mode. See the "31-bit Addressing Mode" section in the chapter on the appropriate operating system.

Figure 81 on page 122 shows a sample VS Pascal program with procedures that can call IMS.

VS Pascal procedure that can be invoked from IMS

```
SEGMENT PASCALIMS;
TYPE
  CHAR2 = PACKED ARRAY [1..2] OF CHAR;
  CHAR4 = PACKED ARRAY [1..4] OF CHAR;
  PCB_REC = RECORD
              DBD_NAME       : ALFA;        (* Database name          *)
              SEG_LEVEL      : CHAR2;       (* Segment level number   *)
              STATUS_CODE    : CHAR2;       (* Status code of call    *)
              PROC_OPTIONS   : CHAR4;       (* Processing options     *)
              RESERVE_DLI    : INTEGER;     (* Reserved for DLI/I     *)
              SEG_NAME_FB    : ALFA;        (* Segment name           *)
              LENGTH_FB_KEY  : INTEGER;     (* Length of Key FB Area  *)
              NUMB_SENS_SEGS : INTEGER;     (* Number of sensitive seg*)
              KEY_FB_AREA    :              (* Key Feedback Area      *)
                 PACKED ARRAY [1..17] OF CHAR;
            END;
PROCEDURE PASCALIMS(VAR SAVE : INTEGER;     (* Addr of Pascal environ.*)
                    VAR DB_PCB : PCB_REC);  (* Addr of DB_PCB         *)
                        REENTRANT;
PROCEDURE PASCALIMS;
TYPE
  IO_REC  = RECORD
              KEY   : PACKED ARRAY [1..n] OF CHAR;
              FIELD : PACKED ARRAY [1..n] OF CHAR;
            END;


  USSA_REC = RECORD
               SEG_NAME : ALFA;             (* Contain segment's name *)
               BLANK    : CHAR;
             END;
VAR
  DLI_FUNC    : CHAR4;                      (* DL/I function code     *)
  DB_PCB      : PCB_REC;                    (* PCB that DL/I will     *)
                                            (* reference in this call *)
  IOAREA      : IO_REC;                     (* Data retrieved or      *)
                                            (* inserted               *)
  UNQUAL_SSA1 : USSA_REC;                   (* Unqualified segment    *)
                                            (* search argument        *)

PROCEDURE PASTDLI; GENERIC;                 (*Declare PASTDLI routine *)

BEGIN
  DLI_FUNC := 'GU  ';                       (* Set DL/I function      *)
  UNQUAL_SSA.SEG_NAME := 'name    ';        (* Set segment name       *)
```

Figure 81 (Part 1 of 2). Example of a VS Pascal Program with Procedures That Can Call and Be Called by IMS

```
  PASTDLI(CONST DLI_FUNC,                  (* DLI function           *)
          VAR   DB_PCB,                    (* PCB                    *)
          VAR   IOAREA,                    (* Information requested  *)
          VAR   UNQUAL_SSA);               (* Segment to be searched *)
END;
```

Figure 81 (Part 2 of 2). Example of a VS Pascal Program with Procedures That Can Call and Be Called by IMS

Part II. Reference

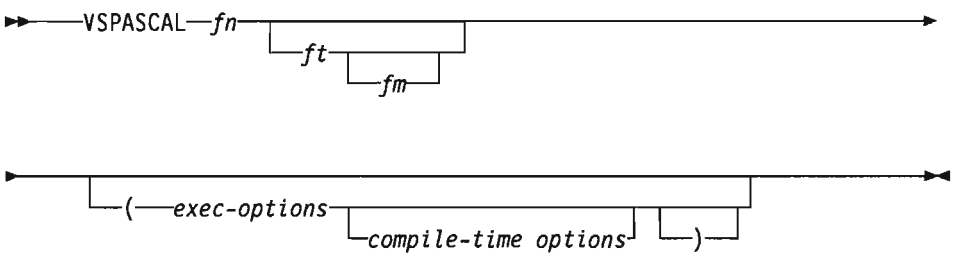
Chapter 10. VM EXECs

Figure 82 summarizes the VS Pascal EXECs in a quick-reference chart. Detailed explanations of each EXEC follow. For more information see Chapter 1, "How to Run a Program under VM" on page 3.

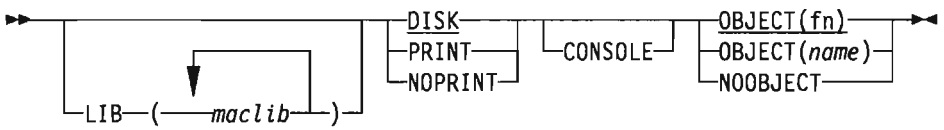
EXEC	Description	See Page
VSPASCAL EXEC	Compiles a VS Pascal source file.	127
PASCMOD EXEC	Link-edits a VS Pascal program.	128
PASCRUN EXEC	Invokes a VS Pascal load module.	130

Figure 82. VS Pascal EXECs

VSPASCAL EXEC



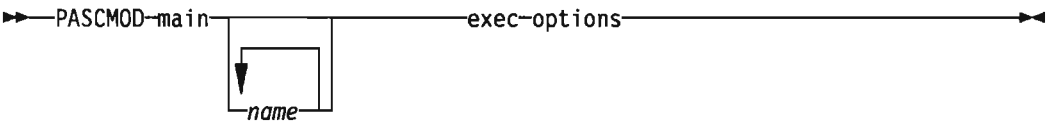
Exec-options



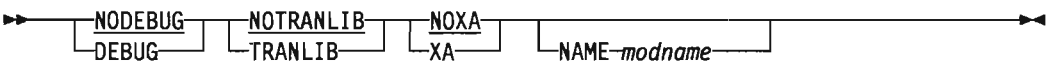
Where	Specifies
VSPASCAL	The command name. **Note:** To access HELP information for this EXEC, enter `VSPASCAL ?`.
fn ft fm	The file name (*fn*), file type (*ft*), and file mode (*fm*) of the source program. The file type and file mode are optional. The default file type is PASCAL, and the default file mode is *.
LIB(*maclibs*)	The macro libraries (*maclibs*) required by the unit being compiled. These MACLIBs contain source code to be inserted by the compiler when it encounters a %INCLUDE compiler directive. For more information on %INCLUDE see "For Programs that Use the %INCLUDE Compiler Directive" on page 3.

	The default MACLIB, named VSPASCAL, need not be specified. It is always implicitly provided as the last MACLIB in the search order.
DISK	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 is the default.
PRINT	The listing is to be spooled to your virtual printer.
NOPRINT	The listing is to be suppressed. This option automatically activates three compile-time options: NOSOURCE, NOXREF, and NOLIST.
CONSOLE	The console messages produced by the compiler are to be stored as a file on your A-disk. The file is named *fn* CONSOLE. When CONSOLE is not specified, then the messages are displayed on your terminal.
OBJECT(*fn*)	The TEXT file is to have the same name (*fn*) as the unit being compiled.
OBJECT(*name*)	The TEXT file is to be named *name*.
NOOBJECT	The production of an object module is suppressed. This option is useful when you want only to check your program for language errors.
compiler-options	The compile-time options. See Chapter 13, "Compile-Time Options" on page 155.

PASCMOD EXEC



Exec-options



Where	**Specifies**
PASCMOD	The command name.
	Note: To access HELP information for this EXEC, enter `PASCMOD ?`.
main	The name of the main program module.
names	The names of segment modules (TEXT decks) and text libraries (TXTLIBs) to be included. If a name is specified and there are two files with the same name (one is named *n* TEXT and the other named *n* TXTLIB), the TEXT file is included *and* the TXTLIB is searched.

XA
: VS Pascal programs can reside above the 16-megabyte line. All program data (both dynamic and static) can be allocated above the 16-megabyte line.

  The CMS command

  ```
  GLOBAL LOADLIB PASRTLIB
  ```

  will be issued. This command makes a load library (LOADLIB) available. This library is required for the XA option.

NOXA
: VS Pascal programs must reside below the 16-megabyte line. This is the default.

DEBUG
: The debugging routines are to be linked into the load module so that the Interactive Debugging Tool can be used.

NODEBUG
: The debugging routines are not to be linked into the load module. This is the default.

TRANLIB
: The module is to be link-edited for execution with the transient run-time library.

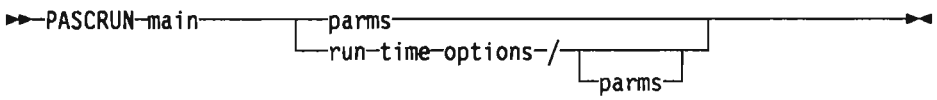
  Transient run-time execution loads all library members needed by a user program when the program begins execution, instead of the library members being link-edited into the source module. Although transient run-time can slow program execution, it decreases the size of the load module and eliminates the need for providing copies of run-time libraries to multiple users. This is particularly helpful to sites that must provide resources to a large number of individual users.

  Remember that because the load module is not fully resolved, it cannot be transported to another site that does not have the IBM-licensed run-time libraries.

NOTRANLIB
: The load module is to be link-edited for standard execution. This is the default.

NAME *modname*
: An alternate name for the load module. The resulting load module and map have the name *name* MODULE A and *name* MAP A.

PASCRUN EXEC



Where	Specifies
PASCRUN	The command name. **Note:** To access HELP information for this EXEC, enter PASCRUN ?.
main	The name of the load module.
run-time options	VS Pascal run-time options. See Chapter 14, "Run-Time Options" on page 167 for more information on the VS Pascal run-time options.
parms	The parameters (if any) being passed to the load module.

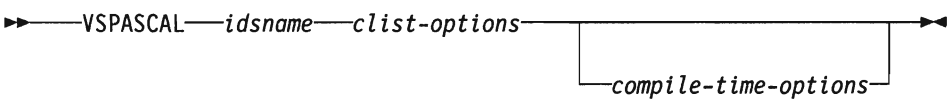
Chapter 11. MVS CLISTs and the CALL Command

Figure 83 summarizes all VS Pascal CLISTs and the MVS/TSO CALL command in a quick-reference chart. Detailed explanations of each command follow. For more information, see Chapter 2, "How to Run a Program under MVS/TSO" on page 13.

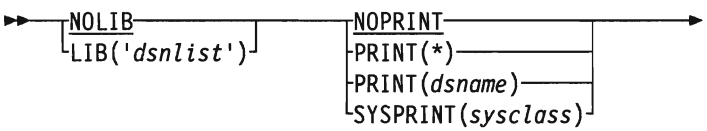
CLIST or Command	Description	See Page
VSPASCAL CLIST	Compiles a VS Pascal source file.	131
PASCMOD CLIST	Link-edits a VS Pascal program.	134
CALL command	Invokes a load module.	136

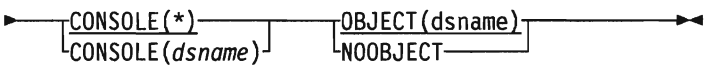
Figure 83. MVS CLISTs and the CALL Command

VSPASCAL CLIST



Clist options





Where	Specifies
VSPASCAL	The command name.
idsname	The name of the primary input data set containing 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 is prefixed and the qualifier 'PASCAL' is suffixed). This must be the first parameter specified.
compile-time options	One or more compile-time options separated by commas or blanks. See Chapter 13, "Compile-Time Options" on page 155.

NOLIB — No %INCLUDE libraries are required. This is the default.

LIB('*dsnlist*') — The list of partitioned data set names (*dsnlist*) that may be specified for use in the %INCLUDE compiler directive. For more information see "For Programs that Use the %INCLUDE Compiler Directive" on page 14.

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 ''...''.

NOPRINT — The listing is to be suppressed. This parameter activates the compile-time options: NOSOURCE, NOXREF, and NOLIST. This is the default.

PRINT(*) — The compiler listing is to be displayed on the terminal; no other copy will be available.

PRINT (*dsname*) — The compiler listing is to be written to the data set named in *dsname*. This can be either a fully qualified name (enclosed within triple quotation marks '''...''') or a simple name (prefixed by the identification qualifier and suffixed by the qualifier 'LIST').

Note: For fully qualified names, the triple quotation marks are required because the CLIST processor removes the outer quotation marks within a keyword suboperand list.

SYSPRINT (*sysclass*)
The compiler listing is to be written to the SYSOUT class named *sysclass*.

CONSOLE(*) — The compiler-generated messages are to be displayed on the terminal. This is the default.

CONSOLE (*dsname*)
The compiler-generated messages are to be written to the data set named (*dsname*). This can be either a fully qualified name (enclosed within triple quotation marks '''...''') or a simple name (prefixed by the identification qualifier and suffixed by the qualifier 'CONSOLE').

Note: For fully qualified names, the triple quotation marks are required because the CLIST processor removes the outer quotation marks within a keyword suboperand list.

OBJECT (*dsname*)

The object module produced by the compiler is to be written to the data set named *dsname*. This can be either a fully qualified name (enclosed within triple quotation marks '''...''') or a simple name (prefixed by the identification qualifier and suffixed by the qualifier 'OBJ').

Note: For fully qualified names, the triple quotation marks are required because the CLIST processor removes the outer quotation marks within a keyword suboperand list.

This is the default.

If neither OBJECT nor NOOBJECT is specified, then the object module produced by the compiler is written to a default data set. If the data set specified in the first operand contains a descriptive qualifier of 'PASCAL', the CLIST forms 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 primary input data set (*idsname*) is a member of a partitioned data set, then the name of the associated object module is generated as just described. If the object module data set is a partitioned data set, then the object module becomes a member within the PDS and has the same name as the member name of the input data set.

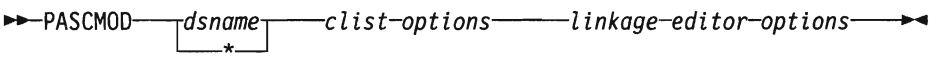
Examples:

- VSPASCAL SORT gives object module 'ABC.SORT.OBJ'
- VSPASCAL 'DEF.PDS.PASCAL(MAIN)' gives object module 'DEF.PDS.OBJ(MAIN)'
- VSPASCAL 'ABC.PROG.PAS' will prompt you for the object module name.

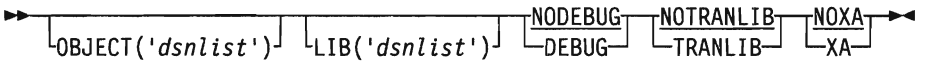
NOOBJECT

No object module is to be produced. The compiler diagnoses errors only.

PASCMOD CLIST



Clist-Options



Where	Specifies
PASCMOD	The command name.
dsname	Specifies the name of a data set containing a VS Pascal object module and/or linkage editor control statements. If more than one object module is to be linked, then their names should appear in the OBJECT parameter. 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('*dsnlist*')	Specifies a list of data sets containing additional object modules to be included in the link-edit. Because of CLIST restrictions, the list must be enclosed in single quotation marks ('...'); fully qualified names within the list must be enclosed in double quotation marks (''....'').
LIB('*dsnlist*')	One or more names of library data sets to be searched by the linkage editor to resolve external references (locate load modules referred to by the module being processed). The name of the VS Pascal run-time 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 quotation marks ('...'); fully qualified names within the list must be enclosed in double quotation marks (''....'').
DEBUG	The debugging routines are to be linked into the load module so that the Interactive Debugging Tool can be used.
NODEBUG	The debugging routines are not to be linked into the load module. This is the default.

TRANLIB The module is to be link-edited for execution with the transient run-time library.

Transient run-time execution loads all library members needed by a user program when the program begins execution, instead of the library members being link-edited into the source module. Although transient run-time can slow program execution, it decreases the size of the load module and eliminates the need for providing copies of run-time libraries to multiple users. This is particularly helpful to sites that must provide resources to a large number of individual users.

Remember that because the load module is not fully resolved, it cannot be transported to another site that does not have the IBM-licensed run-time libraries.

NOTRANLIB

The load module is to be link-edited for standard execution. This is the default.

XA VS Pascal programs can reside above the 16-megabyte line. All program data (both dynamic and static) can be allocated above the 16-megabyte line.

The Interactive Debugging Tool can also be above the 16-megabyte line.

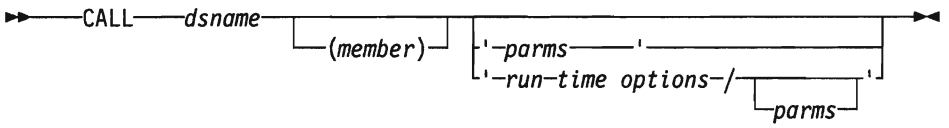
NOXA VS Pascal programs must reside below the 16-megabyte line. This is the default.

linkage-editor-options

Other link-edit options. Refer to *MVS/370 Linkage Editor and Loader User's Guide*, SC26—4061 for information on available options.

The *dsname*, OBJECT, LIB, TRANLIB, XA, and DEBUG parameters are unique to VS Pascal. For more information on the other parameters, see the LINK command in the *TSO Extensions Command Reference*, GC28-1307.

CALL Command



Where	Specifies
CALL	The command name.
dsname (member)	The name of a partitioned data set and the member where the load module to be invoked is stored. If you omit the member name, VS Pascal invokes the member 'TEMPNAME'. *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 quotation marks.
options	One or more run-time options separated by either a comma or a blank. See Chapter 14, "Run-Time Options" on page 167.
parms	A parameter string that is 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.

Chapter 12. MVS Batch Cataloged Procedures

Figure 84 summarizes all MVS Batch cataloged procedures in a quick-reference chart. Detailed explanations of each cataloged procedure follow. For more information see Chapter 3, "How to Run a Program in an MVS Batch Environment" on page 23.

Procedure	Description	See Page
PASCC Procedure	Compiles a source module to produce an object module.	139
PASCCG Procedure	Compiles a source module and executes the resulting load module.	149
PASCCL Procedure	Compiles a source module to produce an object module and then link-edits the object module to produce a load module.	144
PASCCLG Procedure	Compiles, link-edits, produces a load module, then executes the load module.	152
PASCG Procedure	Runs a precompiled program	143
PASCL Procedure	Link-edits a module.	141
PASCLG Procedure	Link-edits, then executes the program.	147

Figure 84. MVS Batch Cataloged Procedures

Data Set Descriptions

Figure 85 describes the data sets needed in VS Pascal cataloged procedures.

DD Statement	In Step Named	Description
SYSIN	PASC	Input data set. You must supply this statement with the qualified ddname PASC.SYSIN.
OUCODE	PASC	Intermediate data produced by the VS Pascal optimizer when the OUCODE compile-time option is specified. For information on the OUCODE compile-time option, see *VS Pascal Diagnosis Guide and Reference*.
OUTPUT	PASC	Output data set for system messages.
STEPLIB	PASC	Data sets containing the VS Pascal compiler modules and run-time environment.
SYSLIB	PASC	Input data sets for %INCLUDE members.
SYSLIN	PASC	Temporary data set for the object module.

Figure 85 (Part 1 of 2). Data Set Descriptions for Cataloged Procedures

DD Statement	In Step Named	Description
SYSLIST	PASC	Data set for pseudo-assembler listing when the LIST compile-time option is specified.
SYSOIN	PASC	Used by VS Pascal for intermediate data.
SYSPRINT	PASC	For listings and diagnostics produced by the compiler.
SYSTERM	PASC	Used for terminal messages.
SYSTIN	PASC	Work data set for VS Pascal intermediate data.
SYSUHDR	PASC	Temporary data set for UHEADERs.
SYSUT1	PASC	Work data set.
SYSUT2	PASC	Work data set.
SYSXREF	PASC	Work data set for cross-reference listing (when XREF compile-time option specified).
UCODE	PASC	Intermediate data produced by the VS Pascal compiler front-end when the UCODE compile-time option is specified. For information on the UCODE compile-time option, see *VS Pascal Diagnosis Guide and Reference*.
SYSIN	LKED	You supply an entry point with this statement when linking multiple modules. The ddname is LKED.SYSIN.
SYSLIB	LKED	Automatic call library.
SYSLIN	LKED	Primary input data set for the linkage editor.
SYSLMOD	LKED	Output load module library.
SYSPRINT	LKED	For diagnostic output messages.
SYSUT1	LKED	Work data set.
STEPLIB	GO	Data set for the run-time environment.
OUTPUT	GO	Output data set for VS Pascal program and system messages.
SYSLIB	GO	Automatic call library.
SYSLIN	GO	Primary input data set for the loader.
SYSLOUT	GO	For loader output.
SYSPRINT	GO	For output messages.

Figure 85 (Part 2 of 2). Data Set Descriptions for Cataloged Procedures

PASCC Procedure

```
//*******************************************************************//
//*                                                                 *//
//*       V S   P A S C A L                                         *//
//*       Version 1, Release 2.0                                    *//
//*       5668-717 (C) Copyright IBM Corp. 1981, 1987, 1988.        *//
//*                                                                 *//
//*       PASCC Procedure:  Compile a VS Pascal program.            *//
//*                                                                 *//
//*       Symbolic Parameters:                                      *//
//*                                                                 *//
//*         HiQual  - High-level qualifier of the VS Pascal datasets *//
//*         MidQual - Mid-level qualifier of the VS Pascal datasets  *//
//*         LoadSet - Dataset to contain generated object code      *//
//*         COpts   - Compiler options                              *//
//*         CLang   - Language for compile-time messages and text   *//
//*         SOut    - SYSOUT                                        *//
//*                                                                 *//
//*******************************************************************//
//PASCC PROC  HIQUAL='VSPASCAL',
//            MIDQUAL='VSPV1R2',
//            LOADSET='PASC.OBJ',
//            COPTS='',
//            CLANG=ENG,
//            SOUT='*'
//*
//*******************************************************************//
//* Compile step.                                                   *//
//*******************************************************************//
//PASC   EXEC PGM=PASCALI,REGION=1M,
//            PARM='LANGUAGE(&CLANG) / &COPTS'
//STEPLIB  DD DSN=&HIQUAL..&MIDQUAL..SAMPMOD1,DISP=SHR
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN2,DISP=SHR[1]
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPMAC1,DISP=SHR
//OUCODE   DD SYSOUT=&SOUT
//OUTPUT   DD SYSOUT=&SOUT
//SYSLIN   DD DSN=&LOADSET,DISP=(NEW,CATLG),
//            SPACE=(TRK,(2,5)),
//            DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB,DSORG=PS)
//SYSLIST  DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(TRK,(2,5))
//SYSOIN   DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(TRK,(2,5))
//SYSPRINT DD SYSOUT=&SOUT,DCB=(RECFM=VBA,LRECL=133,BLKSIZE=685)
//SYSTERM  DD DUMMY
//SYSTIN   DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(TRK,(2,5))
```

Figure 86 (Part 1 of 2). PASCC Procedure

```
//SYSUHDR  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=&SOUT
//*
```

Figure 86 (Part 2 of 2). PASCC Procedure

[1]This is required only if the VS Pascal compiler is installed above the 16-megabyte address line.

PASCL Procedure

```
//*********************************************************************//
//*                                                                   *//
//*       V S   P A S C A L                                           *//
//*       Version 1, Release 2.0                                      *//
//*       5668-717 (C) Copyright IBM Corp. 1988.                      *//
//*                                                                   *//
//*       PASCL Procedure:  Link a VS Pascal program.                 *//
//*                                                                   *//
//*       Symbolic Parameters:                                        *//
//*                                                                   *//
//*         HiQual  - High-level qualifier of the VS Pascal datasets *//
//*         MidQual - Mid-level qualifier of the VS Pascal datasets  *//
//*         LoadSet - Dataset containing previously compiled object  *//
//*                   code                                           *//
//*         GoSet   - Dataset to contain load module                 *//
//*         GoMem   - Member of GoSet to contain load module         *//
//*         LOpts   - Options passed to the Linkage Editor           *//
//*         XA      - 'XA' causes the program to be linked AMODE(31),*//
//*                   RMODE(Any), 'NOXA' causes the program to be    *//
//*                   linked AMODE(31), RMODE(24)                    *//
//*         SOut    - SYSOUT                                         *//
//*                                                                   *//
//*********************************************************************//
//PASCL PROC  HIQUAL='VSPASCAL',
//            MIDQUAL='VSPV1R2',
//            LOADSET='PASC.OBJ',
//            GOSET='PASC.MOD',
//            GOMEM='GO',
//            LOPTS='LIST,MAP',
//            XA='XA',
//            SOUT='*'
//*
//*********************************************************************//
//* Link step.                                                        *//
//*********************************************************************//
//LKED   EXEC PGM=IEWL,PARM='&LOPTS'
//*-------------------------------------------------------------------*//
//* Choose a syslib based on the desired configuration                *//
//*
//* Syslib for transient library (with debugger)
//*
//*SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPDBG2,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN3,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
```

Figure 87 (Part 1 of 2). PASCL Procedure

```
//*
//* Syslib for transient library (without debugger)
//*
//*SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN3,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//*
//* Syslib for normal library (with debugger)
//*
//*SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPDBG1,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//*
//* Syslib for normal library (without debugger)
//*
//SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1,DISP=SHR
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//*
//* End of syslib choices                                              */
//*-------------------------------------------------------------------*//
//SYSLIN   DD DSN=&HIQUAL..&MIDQUAL..SAMPLKD1(&XA),DISP=SHR
//         DD DSN=&LOADSET,DISP=SHR
//         DD DDNAME=SYSIN
//SYSLMOD  DD DSN=&GOSET(&GOMEM),DISP=(NEW,CATLG),
//            SPACE=(TRK,(2,5,1))
//SYSPRINT DD SYSOUT=&SOUT
//SYSUT1   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//*
```

Figure 87 (Part 2 of 2). PASCL Procedure

PASCG Procedure

```
//*********************************************************************//
//*                                                                   *//
//*       V S   P A S C A L                                           *//
//*       Version 1, Release 2.0                                      *//
//*       5668-717 (C) Copyright IBM Corp. 1988.                      *//
//*                                                                   *//
//*       PASCG Procedure:  Execute a VS Pascal program.              *//
//*                                                                   *//
//*       Symbolic Parameters:                                        *//
//*                                                                   *//
//*         HiQual  - High-level qualifier of the VS Pascal datasets *//
//*         MidQual - Mid-level qualifier of the VS Pascal datasets  *//
//*         GoSet   - Dataset containing load module                  *//
//*         GoMem   - Member of GoSet containing load module          *//
//*         ROpts   - Execution options for the VS Pascal program     *//
//*         RLang   - Language for run-time messages                  *//
//*         SOut    - SYSOUT                                          *//
//*                                                                   *//
//*********************************************************************//
//PASCG PROC  HIQUAL='VSPASCAL',
//            MIDQUAL='VSPV1R2',
//            GOSET='PASC.MOD',
//            GOMEM='GO',
//            ROPTS='',
//            RLANG=ENG,
//            SOUT='*'
//*
//*********************************************************************//
//* Execute step.                                                     *//
//*********************************************************************//
//GO     EXEC PGM=&GOMEM,REGION=1M,
//            PARM='LANGUAGE(&RLANG) / &ROPTS'
//STEPLIB  DD DSN=&GOSET,DISP=SHR
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN4,DISP=SHR
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN2,DISP=SHR
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//OUTPUT   DD SYSOUT=&SOUT,DCB=(RECFM=VBA,LRECL=133)
//SYSPRINT DD SYSOUT=&SOUT,DCB=(RECFM=VBA,LRECL=133)
//*
```

Figure 88. PASCG Procedure

PASCCL Procedure

```
//*********************************************************************//
//*                                                                   *//
//*       V S   P A S C A L                                           *//
//*       Version 1, Release 2.0                                      *//
//*       5668-717 (C) Copyright IBM Corp. 1981, 1987, 1988.          *//
//*                                                                   *//
//*       PASCCL Procedure:  Compile and link a VS Pascal program.    *//
//*                                                                   *//
//*       Symbolic Parameters:                                        *//
//*                                                                   *//
//*         HiQual  - High-level qualifier of the VS Pascal datasets  *//
//*         MidQual - Mid-level qualifier of the VS Pascal datasets   *//
//*         GoSet   - Dataset to contain load module                  *//
//*         GoMem   - Member in GoSet for load module                 *//
//*         COpts   - Compiler options                                *//
//*         CLang   - Language for compile-time messages and text     *//
//*         LOpts   - Options passed to the Linkage Editor            *//
//*         XA      - 'XA' causes the program to be linked AMODE(31),*//
//*                   RMODE(Any), 'NOXA' causes the program to be     *//
//*                   linked AMODE(31), RMODE(24)                     *//
//*         SOut    - SYSOUT                                          *//
//*                                                                   *//
//*********************************************************************//
//PASCCL PROC HIQUAL='VSPASCAL',
//            MIDQUAL='VSPV1R2',
//            GOSET='PASC.MOD',
//            GOMEM='GO',
//            COPTS='',
//            CLANG=ENG,
//            LOPTS='LIST,MAP',
//            XA='XA',
//            SOUT='*'
//*
//*********************************************************************//
```

Figure 89 (Part 1 of 3). PASCCL Procedure

```
//* Compile step.                                                      */
//*********************************************************************//
//PASC   EXEC PGM=PASCALI,REGION=1M,
//            PARM='LANGUAGE(&CLANG) / &COPTS'
//STEPLIB  DD DSN=&HIQUAL..&MIDQUAL..SAMPMOD1,DISP=SHR
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN2,DISP=SHR[1]
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//OUCODE   DD SYSOUT=&SOUT
//OUTPUT   DD SYSOUT=&SOUT
//SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPMAC1,DISP=SHR

//SYSLIN   DD DSN=&&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))
//SYSOIN   DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(TRK,(2,5))
//SYSPRINT DD SYSOUT=&SOUT,DCB=(RECFM=VBA,LRECL=133,BLKSIZE=685)
//SYSTERM  DD DUMMY

//SYSTIN   DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(TRK,(2,5))
//SYSUHDR  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=&SOUT
//*

//*********************************************************************//
//* Link step.                                                        */
//*********************************************************************//
//LKED   EXEC PGM=IEWL,COND=(8,LE,PASC),
//            PARM='&LOPTS'
//*-------------------------------------------------------------------*//
//* Choose a syslib based on the desired configuration                */
//*
//* Syslib for transient library (with debugger)
```

Figure 89 (Part 2 of 3). PASCCL Procedure

```
//*
//*SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPDBG2,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN3,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//*
//* Syslib for transient library (without debugger)
//*
//*SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN3,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//*
//* Syslib for normal library (with debugger)
//*
//*SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPDBG1,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//*
//* Syslib for normal library (without debugger)
//*
//SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1,DISP=SHR
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//*
//* End of syslib choices                                              *//
//*-------------------------------------------------------------------*//
//SYSLIN   DD DSN=&HIQUAL..&MIDQUAL..SAMPLKD1(&XA),DISP=SHR
//         DD DSN=*.PASC.SYSLIN,DISP=(OLD,DELETE)
//         DD DDNAME=SYSIN
//SYSLMOD  DD DSN=&GOSET(&GOMEM),DISP=(NEW,CATLG),
//            SPACE=(TRK,(2,5,1))
//SYSPRINT DD SYSOUT=&SOUT
//SYSUT1   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//*
```

Figure 89 (Part 3 of 3). PASCCL Procedure

[1]This is required only if the VS Pascal compiler is installed or executed above the 16-megabyte address line.

PASCLG Procedure

```
//*****************************************************************//
//*                                                               *//
//*       V S   P A S C A L                                       *//
//*       Version 1, Release 2.0                                  *//
//*       5668-717 (C) Copyright IBM Corp. 1981, 1987, 1988.      *//
//*                                                               *//
//*       PASCLG Procedure:  Link and execute a VS Pascal program.*//
//*                                                               *//
//*       Symbolic Parameters:                                    *//
//*                                                               *//
//*         HiQual  - High-level qualifier of the VS Pascal datasets *//
//*         MidQual - Mid-level qualifier of the VS Pascal datasets  *//
//*         LoadSet - Dataset containing previously compiled object  *//
//*                   code                                        *//
//*         LOpts   - Options passed to the Linkage Editor        *//
//*         XA      - 'XA' causes the program to be linked AMODE(31),*//
//*                   RMODE(Any), 'NOXA' causes the program to be  *//
//*                   linked AMODE(31), RMODE(24)                 *//
//*         ROpts   - Execution options for the VS Pascal program *//
//*         RLang   - Language for run-time messages              *//
//*         SOut    - SYSOUT                                      *//
//*                                                               *//
//*****************************************************************//
//PASCLG PROC HIQUAL='VSPASCAL',
//            MIDQUAL='VSPV1R2',
//            LOADSET='PASC.OBJ',
//            LOPTS='LIST,MAP',
//            XA='XA',
//            ROPTS='',
//            RLANG=ENG,
//            SOUT='*'
//*
//*****************************************************************//
//* Link step.                                                    *//
//*****************************************************************//
//LKED   EXEC PGM=IEWL,PARM='&LOPTS'
//*---------------------------------------------------------------*//
//* Choose a syslib based on the desired configuration            *//
//*
//* Syslib for transient library (with debugger)
//*
//*SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPDBG2,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN3,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
```

Figure 90 (Part 1 of 2). PASCLG Procedure

```
//*
//* Syslib for transient library (without debugger)
//*
//*SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN3,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//*
//* Syslib for normal library (with debugger)
//*
//*SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPDBG1,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//*
//* Syslib for normal library (without debugger)
//*
//SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1,DISP=SHR
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//*
//* End of syslib choices                                          *//
//*---------------------------------------------------------------*//
//SYSLIN   DD DSN=&HIQUAL..&MIDQUAL..SAMPLKD1(&XA),DISP=SHR
//         DD DSN=&LOADSET,DISP=SHR
//         DD DDNAME=SYSIN
//SYSLMOD  DD DSN=&&GOSET(GO),UNIT=SYSDA,DISP=(,PASS),
//            SPACE=(TRK,(2,5))
//SYSPRINT DD SYSOUT=&SOUT
//SYSUT1   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//*
//*****************************************************************//
//* Execute step.                                                 *//
//*****************************************************************//
//GO     EXEC PGM=*.LKED.SYSLMOD,REGION=1M,COND=(8,LE,LKED),
//            PARM='LANGUAGE(&RLANG) / &ROPTS'
//STEPLIB  DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN4,DISP=SHR
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN2,DISP=SHR
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//OUTPUT   DD SYSOUT=&SOUT,DCB=(RECFM=VBA,LRECL=133)
//SYSPRINT DD SYSOUT=&SOUT,DCB=(RECFM=VBA,LRECL=133)
//*
```

Figure 90 (Part 2 of 2). PASCLG Procedure

PASCCG Procedure

```
//*********************************************************************//
//*                                                                   *//
//*       V S   P A S C A L                                           *//
//*       Version 1, Release 2.0                                      *//
//*       5668-717 (C) Copyright IBM Corp. 1981, 1987, 1988.          *//
//*                                                                   *//
//*       PASCCG Procedure:  Compile, load, and execute a VS Pascal   *//
//*                          program.                                 *//
//*                                                                   *//
//*       Symbolic Parameters:                                        *//
//*                                                                   *//
//*         HiQual  - High-level qualifier of the VS Pascal datasets *//
//*         MidQual - Mid-level qualifier of the VS Pascal datasets   *//
//*         COpts   - Compiler options                                *//
//*         CLang   - Language for compile-time messages and text     *//
//*         Include - Use 'AMPZMVSB' if program to reside above 16M,  *//
//*                   use 'AMPZRPO1' if program to reside below 16M   *//
//*         Modes   - AMODE and RMODE of program
//*         ROpts   - Execution options for the VS Pascal program     *//
//*         RLang   - Language for run-time messages                  *//
//*         SOut    - SYSOUT                                          *//
//*                                                                   *//
//*********************************************************************//

//PASCCG PROC HIQUAL='VSPASCAL',
//            MIDQUAL='VSPV1R2',
//            COPTS='',
//            CLANG=ENG,
//            INCLUDE='AMPZMVSB',
//            MODES='AMODE=31,RMODE=ANY',
//            ROPTS='',
//            RLANG=ENG,
//            SOUT='*'
//*
//*********************************************************************//

//* Compile step.                                                     *//
//*********************************************************************//
//PASC   EXEC PGM=PASCALI,REGION=1M,
//            PARM='LANGUAGE(&CLANG) / &COPTS'
//STEPLIB  DD DSN=&HIQUAL..&MIDQUAL..SAMPMOD1,DISP=SHR
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN2,DISP=SHR[1]
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPMAC1,DISP=SHR
//OUCODE   DD SYSOUT=&SOUT
//OUTPUT   DD SYSOUT=&SOUT
```

Figure 91 (Part 1 of 3). PASCCG Procedure

```
//SYSLIN   DD DSN=&&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))
//SYSOIN   DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(TRK,(2,5))
//SYSPRINT DD SYSOUT=&SOUT,DCB=(RECFM=VBA,LRECL=133,BLKSIZE=685)
//SYSTERM  DD DUMMY
//SYSTIN   DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(TRK,(2,5))
//SYSUHDR  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=&SOUT
//*
//*******************************************************************//

//* Load and execute step.                                          *//
//*******************************************************************//
//GO     EXEC PGM=LOADER,COND=(8,LE,PASC),
//            PARM='EP=VSPASCAL,&MODES,LET/LANGUAGE(&RLANG)/&ROPTS'
//STEPLIB  DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN4,DISP=SHR
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN2,DISP=SHR
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//*-----------------------------------------------------------------*//
//* Choose a syslib based on the desired configuration              *//
//*

//* Syslib for transient library (with debugger)
//*
//*SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPDBG2,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN3,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//*
```

Figure 91 (Part 2 of 3). PASCCG Procedure

```
//* Syslib for transient library (without debugger)
//*
//*SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN3,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//*
//* Syslib for normal library (with debugger)
//*
//*SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPDBG1,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//*
//* Syslib for normal library (without debugger)
//*
//SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1,DISP=SHR
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//*
//* End of syslib choices                                          *//
//*-----------------------------------------------------------------*//
//OUTPUT   DD SYSOUT=&SOUT,DCB=(RECFM=VBA,LRECL=133)
//SYSLIN   DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1(&INCLUDE),DISP=SHR
//         DD DSN=*.PASC.SYSLIN,DISP=(OLD,DELETE)
//SYSLOUT  DD SYSOUT=&SOUT
//SYSPRINT DD SYSOUT=&SOUT,DCB=(RECFM=VBA,LRECL=133)
//*
```

Figure 91 (Part 3 of 3). PASCCG Procedure

[1]This is required only if the VS Pascal compiler is installed or executed above the 16-megabyte line.

PASCCLG Procedure

```
//*********************************************************************//
//*                                                                   *//
//*       V S   P A S C A L                                           *//
//*       Version 1, Release 2.0                                      *//
//*       5668-717 (C) Copyright IBM Corp. 1981, 1987, 1988.          *//
//*                                                                   *//
//*       PASCCLG Procedure:  Compile, link, and execute a VS Pascal *//
//*                           program.                                *//
//*                                                                   *//
//*       Symbolic Parameters:                                        *//
//*                                                                   *//
//*         HiQual  - High-level qualifier of the VS Pascal datasets *//
//*         MidQual - Mid-level qualifier of the VS Pascal datasets  *//
//*         COpts   - Compiler options                                *//
//*         CLang   - Language for compile-time messages and text     *//
//*         LOpts   - Options passed to the Linkage Editor            *//
//*         XA      - 'XA' causes the program to be linked AMODE(31),*//
//*                   RMODE(Any), 'NOXA' causes the program to be     *//
//*                   linked AMODE(31), RMODE(24)                     *//
//*         ROpts   - Execution options for the VS Pascal program     *//
//*         RLang   - Language for run-time messages                  *//
//*         SOut    - SYSOUT                                          *//
//*                                                                   *//
//*********************************************************************//
//PASCCLG PROC HIQUAL='VSPASCAL',
//            MIDQUAL='VSPV1R2',
//            COPTS='',
//            CLANG=ENG,
//            LOPTS='LIST,MAP',
//            XA='XA',
//            ROPTS='',
//            RLANG=ENG,
//            SOUT='*'
//*
//*********************************************************************//
//* Compile step.                                                     *//
//*********************************************************************//
//PASC   EXEC PGM=PASCALI,REGION=1M,
//            PARM='LANGUAGE(&CLANG) / &COPTS'
//STEPLIB  DD DSN=&HIQUAL..&MIDQUAL..SAMPMOD1,DISP=SHR
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN2,DISP=SHR[1]
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//OUCODE   DD SYSOUT=&SOUT
//OUTPUT   DD SYSOUT=&SOUT
```

Figure 92 (Part 1 of 3). PASCCLG Procedure

```
//SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPMAC1,DISP=SHR
//SYSLIN   DD DSN=&&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))
//SYSOIN   DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(TRK,(2,5))
//SYSPRINT DD SYSOUT=&SOUT,DCB=(RECFM=VBA,LRECL=133,BLKSIZE=685)
//SYSTERM  DD DUMMY
//SYSTIN   DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(TRK,(2,5))
//SYSUHDR  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=&SOUT
//*
//*********************************************************************//
//* Link step.                                                        *//
//*********************************************************************//
//LKED   EXEC PGM=IEWL,COND=(8,LE,PASC),
//            PARM='&LOPTS'
//*-------------------------------------------------------------------*//
//* Choose a syslib based on the desired configuration                *//
//*
//* Syslib for transient library (with debugger)
//*
//*SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPDBG2,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN3,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//*
//* Syslib for transient library (without debugger)
//*
//*SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN3,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//*
//* Syslib for normal library (with debugger)
//*
//*SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPDBG1,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1,DISP=SHR
//*         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//*
```

Figure 92 (Part 2 of 3). PASCCLG Procedure

```
//* Syslib for normal library (without debugger)
//*
//SYSLIB   DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN1,DISP=SHR
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//*
//* End of syslib choices                                          *//
//*----------------------------------------------------------------*//
//SYSLIN   DD DSN=&HIQUAL..&MIDQUAL..SAMPLKD1(&XA),DISP=SHR
//         DD DSN=*.PASC.SYSLIN,DISP=(OLD,DELETE)
//         DD DDNAME=SYSIN
//SYSLMOD  DD DSN=&&GOSET(GO),UNIT=SYSDA,DISP=(,PASS),
//            SPACE=(TRK,(2,5,1))
//SYSPRINT DD SYSOUT=&SOUT
//SYSUT1   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//*
//*****************************************************************//
//* Execute step.                                                 *//
//*****************************************************************//
//GO     EXEC PGM=*.LKED.SYSLMOD,REGION=1M,
//            COND=((8,LE,PASC),(8,LE,LKED)),
//            PARM='LANGUAGE(&RLANG) / &ROPTS'
//STEPLIB  DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN4,DISP=SHR
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPRUN2,DISP=SHR
//         DD DSN=&HIQUAL..&MIDQUAL..SAMPMSG1,DISP=SHR
//OUTPUT   DD SYSOUT=&SOUT,DCB=(RECFM=VBA,LRECL=133)
//SYSPRINT DD SYSOUT=&SOUT,DCB=(RECFM=VBA,LRECL=133)
//*
```

Figure 92 (Part 3 of 3). PASCCLG Procedure

[1]This is required only if your program will execute above the 16-megabyte address line.

Chapter 13. Compile-Time Options

Figure 93 summarizes all VS Pascal compile-time options in a quick-reference chart. Detailed explanations of each option follow in alphabetic order, starting on page 157. All compile-time options are IBM extensions to Standard Pascal.

Compile-Time Option	Abbreviated Name	Default	Function	See Page
CHECK NOCHECK	—	CHECK	Enables or disables run-time error checking.	157
CONDPARM(*varname1* ='*string*'[, *varname2* ='string']..)	—	—	Controls when selected sections of source are compiled.	159
DDNAME(COMPAT) DDNAME(UNIQUE)	—	DDNAME(COMPAT)	Controls how VS Pascal generates ddnames for files.	160
DEBUG NODEBUG	—	NODEBUG	Controls whether the compiler prepares the unit for debugging with the Interactive Debugging Tool.	161
FLAG(I|W|E|S)	—	FLAG(I)	Controls which messages (informational, warning, error, or severe error) are listed.	161
GOSTMT NOGOSTMT	GS NOGS	GOSTMT	Controls whether a statement table is included within the object code.	161
GRAPHIC NOGRAPHIC	—	NOGRAPHIC	Controls whether the compiler recognizes the shift-out (X'0E') and shift-in (X'0F') characters as bracketing double-byte characters (DBCS characters) in string literals, comments, and the %TITLE and %WRITE compiler directives.	162
HEADER NOHEADER	—	HEADER	Controls whether a header appears above the generated code of each routine.	162
LANGLVL(ANSI83) LANGLVL(EXTENDED)	—	LANGLVL(EXTENDED)	Controls whether the compiler accepts Standard Pascal or full VS Pascal.	162
LANGUAGE(*ccc*)	—	The language specified at installation.	Specifies a language other than the default language for textual information.	163
LINECOUNT(*n*)	LC(*n*)	LINECOUNT(60)	Specifies the number of lines to appear on each page of the output listing.	163

Figure 93 (Part 1 of 2). Summary of Compile-Time Options

Compile-Time Option	Abbreviated Name	Default	Function	See Page
LIST NOLIST	—	NOLIST	Controls the generation of the pseudo-assembler listing.	163
MARGINS(*m*,*n*)	MAR(*m*,*n*)	MARGINS(1,72)	Sets the left and right margins of the input program.	164
OPTIMIZE NOOPTIMIZE	OPT NOOPT	OPTIMIZE	Controls whether the compiler generates optimized code.	164
PAGEWIDTH(*n*)	PW(*n*)	PAGEWIDTH(128)	Specifies the maximum number of characters that can appear on a single line of the output listing.	164
PXREF NOPXREF	—	PXREF	Specifies that the right margin of the output listing is to contain cross-reference entries.	164
SEQUENCE(*m*,*n*) NOSEQUENCE	SEQ(*m*,*n*) NOSEQ	SEQUENCE(73,80)	Specifies which columns within the program being compiled are reserved for a sequence field.	165
SOURCE NOSOURCE	S NOS	SOURCE	Controls the generation of the compiler source listing.	165
STDFLAG(I|W|E|S)	—	STDFLAG(E) for LANGLVL(ANSI83)	Controls how most standard extensions are flagged (informational, warning, error, or severe error message) when LANGLVL(EXTENDED) is not in use.	165
WRITE NOWRITE	—	NOWRITE	Controls whether messages in a %WRITE compiler directive are written to the terminal during compilation.	166
XREF(SHORT) XREF(LONG) NOXREF	X NOX	XREF(SHORT)	Controls the generation of the cross-reference portion of the source listing.	166

Figure 93 (Part 2 of 2). Summary of Compile-Time Options

CHECK Option

CHECK causes in-line run-time error checking to be generated. If NOCHECK is specified, all run-time checking will be suppressed.

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

Default: CHECK.

The run-time 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.

String truncation
Values assigned to string variables are checked to ensure that the variable receiving the value is large enough to contain the value.

This checking may occur:

- When a string variable appears on the left side of an assignment statement.
- When a string expression is passed by value or CONST to a routine.
- When a string variable is passed by VAR to a routine.

Subrange bounds
Variables declared as subrange bounds 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 a subrange variable or function result appears on the left side of an assignment statement.
- When a value is passed by value or CONST to a formal subrange parameter in a routine (including calls to the WRITE procedure for record files).
- After a subrange variable is passed by VAR to a routine (including the READ, READLN, and READSTR procedures).
- When certain predefined routines are used (such as CHR) that require their arguments to be in a certain range.

For the sake of efficiency, the compiler may suppress checking when it can 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;
  .
  .
BEGIN;
  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   *)
END;
```

The compiler makes no explicit attempt to diagnose the use of uninitialized variables. However, to detect such errors, the SETMEM run-time option can be used (see Chapter 14, "Run-Time Options" on page 167).

Subscript ranges

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

When a run-time checking error occurs, a diagnostic message will be displayed on your terminal followed by a trace back report of the routines which were active when the error occurred. If the program is invoked from batch mode, the diagnostic message and trace-back report will be sent to the data set or device associated with ddname SYSPRINT. You can use the "ERRFILE" option (see Chapter 14, "Run-Time Options" on page 167) to direct the error diagnostics to any file you choose.

See "Reading a VS Pascal Trace-Back Report" on page 73 for an example of a trace-back report due to a checking error.

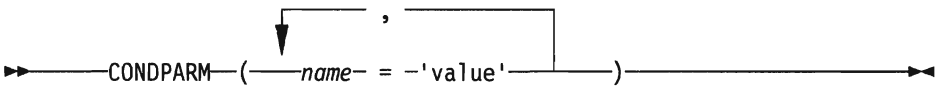
"How You Handle Run-Time Errors with the ONERROR Procedure" on page 78 describes how checking errors may be intercepted by your program.

CONDPARM Option

CONDPARM allows you to compile only selected sections of source code. CONDPARM passes a conditional parameter to the compiler, which evaluates the Boolean expressions used in %WHEN compiler directives.

In the source code, one or more %WHEN statements are enclosed by the %SELECT and %ENDSELECT compiler directives. VS Pascal will compile code enclosed by %SELECT and %ENDSELECT only when the Boolean expression in a %WHEN statement evaluates true. The compiler compiles from the start of that %WHEN statement until it reaches another %WHEN or the %ENDSELECT. For more information on the %SELECT, %WHEN, and %ENDSELECT compiler directives, see *VS Pascal Language Reference*.

The form of the CONDPARM option is:



Where	Represents
name	The name of the conditional Parameter. This parameter name must follow VS Pascal identifier rules and must not be more than 16 characters.
'value'	The character string value associated with the specified name. The conditional parameter value must: • Be enclosed in single quotes • Not exceed 16 characters.

For example

```
CONDPARM ( PARM1 = 'DOG' , PARM2 = 'CAT' )
```

will set the value of PARM1 to DOG and the value of PARM2 to CAT.

Note: When using the CONDPARM option under VM/SP or VM/XA, parameter list tokenization into eight character segments prohibits conditional parameter names longer than eight characters and values longer than six characters. Also, while in a VM environment, blanks should be inserted as in the example above. The value will be folded to upper case by the operating system.

Figure 94 illustrates how to use CONDPARM for conditional compilation.

```
PROGRAM ABSOLUTEZERO(OUTPUT);
VAR
  TEMPERATURE : REAL;

BEGIN
  TEMPERATURE := -459.40;

  %SELECT
    %WHEN (FORM = 'CLSIUS')
      TEMPERATURE = (TEMPERATURE - 32) * 5/9;
    %WHEN (FORM = 'KELVIN')
      TEMPERATURE = ( (TEMPERATURE - 32) * 5/9) + 273;
  %ENDSELECT

  WRITELN(TEMPERATURE:4:2);
END.
```

If you enter

```
CONDPARM(FORM = 'CLSIUS')
```

then AbsoluteZero outputs -273.00.

If you enter

```
CONDPARM(FORM = 'KELVIN')
```

then AbsoluteZero outputs 0.00.

Otherwise, AbsoluteZero outputs -459.40.

Figure 94. Example of Using CONDPARM for Conditional Compilation

DDNAME Option

DDNAME controls how ddnames for files are generated.

COMPAT instructs the compiler to use Pascal/VS Release 2.2 conventions to generate ddnames. Using these conventions, COMPAT generates ddnames from the first eight characters in a file variable's name.

UNIQUE generates unique ddnames for each file variable to conform to Standard Pascal scoping rules. See Figure 27 on page 55 for more information on how ddnames are generated.

Default: DDNAME(COMPAT).

DEBUG Option

DEBUG indicates that the compiler is to produce the information that the Interactive Debugging Tool needs in order to debug VS Pascal programs. The Interactive Debugging Tool is described in Chapter 15, "Interactive Debugging Tool Commands" on page 173.

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

NODEBUG indicates that the Interactive Debugging Tool cannot be used for this unit.

Default: NODEBUG.

FLAG Option

FLAG controls the lowest class of message listed. The class specified *and* any message of a higher class are listed.

I List informational messages. This is the default, which means all messages will be listed.

W List warning messages.

E List error messages.

S List severe error messages.

Default: FLAG(I).

GOSTMT Option

GOSTMT 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 debugging tool to place breakpoints based on source statement numbers. For a description of the Interactive Debugging Tool, see Chapter 15, "Interactive Debugging Tool Commands" on page 173.

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.

Default: GOSTMT.

GRAPHIC Option

GRAPHIC indicates that the VS Pascal compiler will treat the shift-out (X'0E') and shift-in (X'0F') characters as bracketing double-byte characters (DBCS characters) in string literals, comments, and the %TITLE and %WRITE compiler directives. Every shift-out character must have a shift-in character on the same line. The NOGRAPHIC compile-time option implies that shift-out and shift-in characters have no special meaning.

This option only affects the compilation of programs and will have no effect on program execution.

Default: NOGRAPHIC.

HEADER Option

HEADER places a header after the generated code of each routine. This header contains: (1) the name of the compiled unit; (2) the compiler name; (3) the VS Pascal version/release number; and (4) the date and time the unit was compiled. You can also place your own customized header after the compiler-generated header by using the %UHEADER compiler directive in the source code. For more information on the %UHEADER compiler directive, see *VS Pascal Language Reference*.

Default: HEADER.

Note: HEADER information can be viewed only in the object code.

LANGLVL Option

LANGLVL(ANSI83) indicates that the compiler is to diagnose all constructs and features which do not conform to the ANSI standard. Violations of the standard will appear as compiler errors. VS Pascal extensions will not be supported and non-ANSI Standard Pascal reserved words will not be recognized.

If LANGLVL(EXTENDED) is specified, the full VS Pascal language is to be supported.

Default: LANGLVL(EXTENDED).

LANGUAGE Option

LANGUAGE specifies the language used for run-time messages, report headings, and other textual information presented by VS Pascal. The LANGUAGE compile-time option allows you to specify another language for textual information, using a three-character identifier for the new language you desire.

The language is specified by LANGUAGE(*ccc*), where *ccc* is one of the following:

Where	Represents
ENG	Mixed-case English.
UEN	Uppercase English.
JPN	Japanese.

Default: The default language specified during installation.

LINECOUNT Option

LINECOUNT 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.

Minimum: 13.

Default: 60.

LIST Option

LIST and NOLIST control the generation or suppression of the translator pseudo-assembler listing (see "Assembler Listing" on page 40).

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

Default: NOLIST.

MARGINS Option

MARGINS(m,n) 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.

Range: 1 through 100
Default: MARGINS(1,72).

Notes:

1. The specified margins must not overlap the sequence field.
2. When the VSPASCAL CLIST is invoked under TSO, the subparameters of the MARGINS option must be enclosed in quotes. For example,

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

OPTIMIZE Option

OPTIMIZE 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 sub-expressions. For more information on common sub-expression elimination, see "Common Subexpression Elimination" on page 210 for more information.

Default: OPTIMIZE.

For information on other VS Pascal optimizations, see Chapter 19, "Performance Considerations" on page 205.

PAGEWIDTH Option

PAGEWIDTH specifies the maximum number of characters (not including carriage control characters) which may appear on a single line of the output listing. This number depends on the page form and the printer model.

Range: 120 through 210
Default: PAGEWIDTH(128).

PXREF Option

PXREF specifies that the right margin of the output listing is to contain cross reference entries (see page cross-reference field in "Source Listing" on page 34). NOPXREF suppresses these entries.

Default: PXREF.

SEQUENCE Option

SEQUENCE(*m*,*n*) 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.

Range: 1 through 100
Default: SEQ(73,80).

Notes:

1. The sequence field must not overlap the source margins.
2. When the VSPASCAL CLIST is invoked under TSO, the subparameters of the SEQ option must be enclosed in quotes. For example,

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

SOURCE Option

SOURCE and NOSOURCE control 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.

Default: SOURCE.

STDFLAG Option

STDFLAG controls how most standard extensions are flagged when not using LANGLVL(EXTENDED):

I Informational message.

W Warning message.

E Error message. This is the default if LANGLVL(ANSI83) is specified.

S Severe message.

This option has no effect for LANGLVL(EXTENDED).

WRITE Option

WRITE allows messages in a %WRITE compiler directive to be written to the terminal at a specified location in the program during compilation. NOWRITE does not allow messages to be written to the terminal.

Default: NOWRITE.

XREF Option

XREF and NOXREF control the generation or suppression of the cross-reference portion of the source listing. (See "Cross-Reference Listing" on page 36.)

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 referenced.

To specify a particular listing mode, either the word LONG or SHORT is placed after the XREF specification and enclosed within parentheses. For example, the specification

```
XREF(LONG)
```

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

Default: XREF(SHORT).

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

Chapter 14. Run-Time Options

Figure 95 summarizes all VS Pascal run-time options in a quick-reference chart. Detailed explanations of each option follow in alphabetic order, starting on page 168.

Note: When you invoke a VS Pascal program, always specify any run-time options first, and end the option list with a slash ("/"). Follow the slash with any additional parameter string that the program must process.

Run-Time Option	Default	Function	See Page
COUNT	—	Instructs VS Pascal to collect instruction frequency information.	168
DEBUG DEBUG(PROMPT) DEBUG(NOPROMPT)	DEBUG(PROMPT)	Controls when and how the Interactive Debugging Tool gains control during program execution.	168
ERRCOUNT(*n*)	ERRCOUNT(20)	Specifies how many non-fatal errors can occur before the program is abnormally terminated.	168
ERRFILE(*ddname*)	**VM/CMS** Terminal **MVS/TSO** Terminal **MVS Batch** SYSPRINT	Specifies the ddname of the file to which all run-time diagnostics, counting information, and debugger output is written.	168
HEAP(*initsize,incrsize*) HEAP(*incrsize*)	HEAP(12,12) HEAP(12)	Controls how VS Pascal creates and maintains heaps.	169
LANGUAGE(*ccc*)	The language specified at installation.	Specifies a language other than the default language for textual information.	170
MAINT	—	Specifies that trace backs include a list of active run-time routines when an error occurs.	170
NOCHECK	—	Instructs VS Pascal to ignore checking errors during program execution.	170
NOSPIE	—	Instructs VS Pascal not to issue a SPIE request, preventing interception of program interrupts.	170
SETMEM	—	Specifies that, upon entry to each VS Pascal routine, each byte of storage in which the routine's local variables are allocated will be set to a specific hexadecimal value, namely X'FE'.	170

Figure 95 (Part 1 of 2). Summary of Run-Time Options

Run-Time Option	Default	Function	See Page
STACK(*n*)	STACK(12)	Specifies the number of kilobytes that the run-time stack is extended each time the stack overflows.	171

Figure 95 (Part 2 of 2). Summary of Run-Time Options

COUNT Option

COUNT specifies that instruction frequency information is to be collected during program execution. After the program is completed, this information is written to the file specified by ERRFILE. This option will only have an effect if the program was both compiled and link-edited with the DEBUG option.

Note: This option causes a small degradation in performance.

DEBUG Option

DEBUG or DEBUG(PROMPT) specifies that the Interactive Debugging Tool is to gain initial control when you invoke your program. This option is valid only if the load module was generated with the DEBUG option.

DEBUG(NOPROMPT) specifies that the Interactive Debugging Tool will be active but will prompt you for a debugging command only when a run-time error is detected.

Default: DEBUG(PROMPT)

Notes:

1. This option causes a small degradation in performance.
2. Specifying DEBUG causes SETMEM to be activated.
3. The NOSPIE option should not be used with the DEBUG option (breakpoints and statement walking will not work).

ERRCOUNT Option

ERRCOUNT(*n*) specifies how many non-fatal errors (*n*) can occur before the program is abnormally terminated.

Default: ERRCOUNT(20)

ERRFILE Option

ERRFILE(*ddname*) specifies the ddname of the file to which all run-time diagnostics, counting information, and debugger information are written. Under CMS and TSO, this information is displayed on your terminal by default. Under MVS batch, the default error file is SYSPRINT.

HEAP Option

HEAP controls the way VS Pascal creates and maintains heaps. Heaps are areas of storage from which VS Pascal allocates memory for dynamic variables. A heap is created by a call to the procedure NEWHEAP; the NEW and MARK procedures may also create a new heap if there is no active heap.

HEAP takes two forms:

HEAP(*initsize,incrsize*)
HEAP(*incrsize*)

Where	**Specifies**
initsize	The initial size of each new heap in kilobytes.
incrsize	How many kilobytes a heap is extended on overflow.

Note: Both values must be positive, nonzero integers.

Default: HEAP(12,12)

Notes:

1. When VS Pascal exhausts a heap, it issues a GETMAIN to allocate more storage for the heap. If the memory space required by NEW is greater than *incrsize*, GETMAIN allocates the amount of needed space, rounded up to the nearest kilobyte.
2. There is a significant overhead penalty for each invocation of GETMAIN. If you underestimate *incrsize*, GETMAIN is invoked frequently, adversely affecting execution speed. If you overestimate *incrsize*, the heap sets aside storage that is never used. When estimating heap requirements, remember that the two HEAP parameters control all user heaps, and the I/O heap.

For more information, see Chapter 18, "Managing Storage" on page 199.

LANGUAGE Option

LANGUAGE specifies the language used for run-time messages, report headings, and other textual information presented by VS Pascal. The LANGUAGE run-time option allows you to specify another language for textual information, using a three-character identifier for the new language you desire.

The language is specified by LANGUAGE(*ccc*), where *ccc* is one of the following:

Where	Represents
ENG	Mixed-case English.
UEN	Uppercase English.
JPN	Japanese.

Default: The default language specified during installation.

MAINT Option

MAINT specifies that when a run-time error occurs, the trace back is to list active run-time 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 Option

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

NOSPIE Option

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

Note: The Interactive Debugging Tool uses SPIE for breakpoints, and therefore NOSPIE should not be used with the DEBUG option.

SETMEM Option

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

Note: This option causes a small degradation in performance.

STACK Option

STACK(*n*) specifies the number of kilobytes (*n*) 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.

Notes:

1. When the end of the stack is reached, the GETMAIN supervisor call is invoked to allocate more storage for the stack. If the length of the DSA required is greater than *n*, the amount allocated will be the length of the DSA rounded up to the next kilobyte.
2. 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 storage than is necessary.

Default: STACK(12).

Chapter 15. Interactive Debugging Tool Commands

Figure 96 summarizes all VS Pascal commands used with the Interactive Debugging Tool. For more information on debugging, see Chapter 7, "How to Debug Your Program" on page 83. Except for QUIT, END, and CLEAR, you can abbreviate all commands to one letter. Use semicolons to separate multiple commands on a line.

All output produced by the debugger (except for program tracing) will go to the file specified by ERRFILE. If ERRFILE is not specified, the output goes to the terminal in VM/CMS and MVS/TSO, and to SYSPRINT in MVS Batch.

All input to the debugger comes from the terminal in VM/CMS and MVS/TSO, and from SYSIN in MVS Batch.

Debugging Command	Abbreviation	Function	See Page
BREAK	B	Sets a breakpoint at a specified statement.	174
CLEAR	—	Removes all breakpoints.	176
CMS	C	Activates the CMS subset mode.	176
DISPLAY	D	Displays information about the current debugging session.	177
DISPLAY BREAKS	D B	Produces a list of all current breakpoints.	177
DISPLAY COUNTS	D C	Displays instruction frequency information.	177
DISPLAY EQUATES	D E	Produces a list of all equate symbols and their current definitions.	178
END	—	Terminates the program immediately.	178
EQUATE	E	Equates an identifier name to a data string.	178
GO	G	Starts or resumes program execution.	179
HELP or ?	H or ?	Lists all the debugging commands.	179
LISTVARS	L	Displays the values of all variables that are local to the currently qualified routine.	180
QUAL	Q	Sets the scope of variables to be displayed.	180
QUIT	—	Terminates the program immediately.	180
RESET	R	Removes a specified breakpoint.	181
SET ATTR	S A	Sets the default method for viewing variables.	181
SET COUNT	S C	Initiates and terminates statement counting.	182
SET TRACE	S T	Activates or deactivates program tracing.	182
TRACE	T	Produces a routine trace-back.	183

Figure 96 (Part 1 of 2). Summary of Debugging Commands

Debugging Command	Abbreviation	Function	See Page
view variable	*,variable*	Shows the user the contents of a variable during program execution.	183
view storage	*,hex-string*	Displays the contents of a specific storage location.	183
WALK	W	Executes the next statement and then halts the program.	184

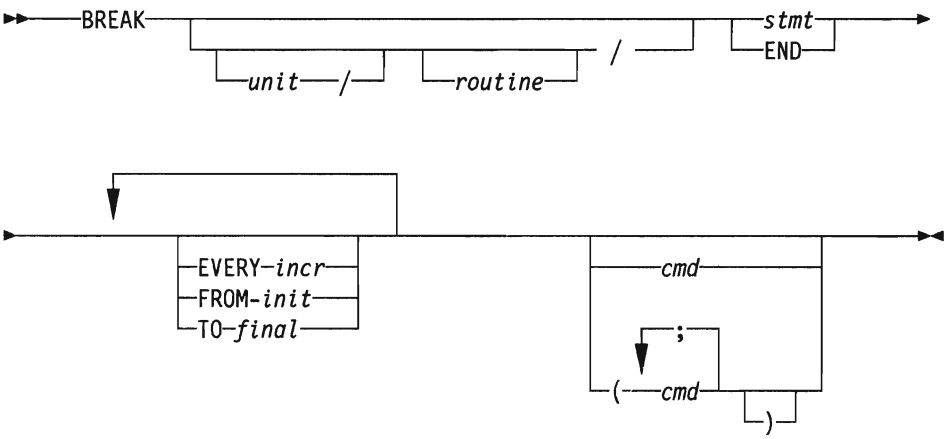
Figure 96 (Part 2 of 2). Summary of Debugging Commands

BREAK Command

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

If you have two or more procedures with the same name in a module, the breakpoint setting may be unpredictable.

A maximum of 32 breakpoints may be set at any one time.



Where **Specifies**

BREAK The command keyword.
Minimum abbreviation: B

unit The name of a VS Pascal unit.

routine The name of a procedure or function in the unit.

stmt The number of the statement at which to stop in the specified routine of the specified unit. The statement numbers are found on the source listing.

EVERY, FROM, TO
The way in which the Interactive Debugging Tool steps through repeated executions of a statement.

- EVERY determines how many repetitions of the statement must occur (*incr*) between breakpoints.
  Default: EVERY 1

- FROM determines at which repetition of the statement the first (*init*) breakpoint occurs.
  Default: FROM 0
- TO determines at which repetition of the statement the last (*final*) breakpoint occurs. *Final* must be greater than or equal to *init*.
  Default: TO MAXINT

Notes:

1. The parameter *init* must be an integer greater than or equal to zero, and the parameters *incr* and *final* must be positive integers greater than zero.
2. If EVERY, FROM, or TO is specified more than once, the last specification of the command will be used.
3. EVERY, FROM, and TO may be specified in any order.

END The breakpoint is to occur in the epilog of the routine before the routine's return.

cmd The action to be taken upon encountering a breakpoint. If you omit the right parenthesis, the rest of the line will be considered part of the associated command string.

Note: The Interactive Debugging Tool executes commands on a BREAK only when a breakpoint actually occurs.

The *unit* and/or *routine* may be omitted in which case the defaults are taken from the current qualification.

The following table illustrates the meaning of the various forms.

Input	**Unit**	**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	**Specifies**
current	The currently qualified unit or procedure.
M,P	The names of a unit or procedure.
S	Either a statement number or END.

Examples

```
B 3
```

will stop at statement 3 of the current routine in the current unit.

```
B P/2 LISTVARS
```

will stop at statement 2 in routine P in the current unit and execute the LISTVARS command.

```
BREAK 12 EVERY 2
```

will stop at every second execution of statement 12.

```
B M/P/END (LISTVARS;GO)
```

will execute the LISTVARS command at the end of routine P in unit M and resume execution.

```
BREAK 12 EVERY 4 FROM 3 TO 16 LISTVARS
```

will stop at the third, seventh, eleventh, and fifteenth occurrences of statement 12, and executes the LISTVARS command at every instance of the breakpoint.

CLEAR Command

CLEAR removes all breakpoints.

Minimum abbreviation: CLEAR

CMS Command

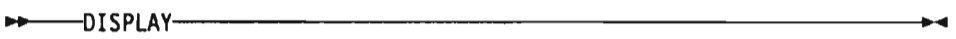
CMS activates the CMS subset mode. If the program is not being run under CMS, the command is ignored. Typing "RETURN" returns to the debug mode from CMS subset mode.

Minimum abbreviation: C

DISPLAY Command

DISPLAY displays information about the current debugging session. The information displayed is:

- The current qualification
- Where the user's program will resume execution after a GO or WALK command
- The current status of counts
- The current status of tracing.



Minimum abbreviation: D

DISPLAY BREAKS Command

DISPLAY BREAKS displays information about breakpoints which are currently set. This information includes:

- Break number
- Break unit
- Break routine
- Break statement
- Break command (if any)
- EVERY, FROM, and TO values (if any)
- Times the statement has executed if EVERY, FROM, or TO is specified.

Minimum abbreviation: D B

DISPLAY COUNTS Command

DISPLAY COUNTS displays instruction frequency information on your terminal.

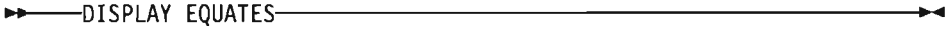
Minimum abbreviation: D C

Notes:

1. You must use the SET COUNT ON debugging command or the COUNT run-time option to activate statement counting.
2. Counting is done on a "basic block" basis. A basic block is a group of statements entered only from the first statement. This implies that a line can be counted as executed if any line in the basic block has been executed.

DISPLAY EQUATES Command

DISPLAY EQUATES produces a list of all equate symbols and their current definitions.



Minimum abbreviation: D E

END Command

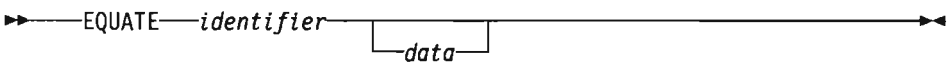
END terminates the program immediately. This command is synonymous with QUIT.

Minimum abbreviation: END

EQUATE Command

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

A maximum of 12 EQUATES is allowed.



Where	Specifies
EQUATE	The command keyword. *Minimum abbreviation*: E
identifier	A VS Pascal identifier.
data	A command that the *identifier* is to represent.

Examples

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

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

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

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

A semicolon will 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 debugging command. There is one exception: the name EQUATE may not be equated to a data string.

```
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
```

removes the symbol "Z" from the Interactive Debugging Tool's equate table.

Equates may be equated to strings which contain other equates. The commands:

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

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

GO Command

GO starts or resumes program execution. The program will continue to execute until one of the following events occurs:

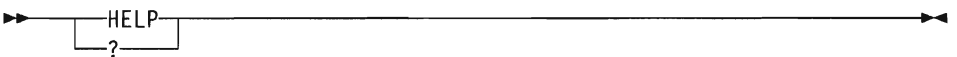
- Breakpoint
- Program error
- Normal program exit

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

Minimum abbreviation: G

HELP Command

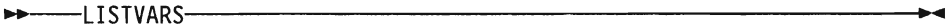
HELP lists all the debugging commands.



Minimum abbreviation: H or ?

LISTVARS Command

LISTVARS displays the values of all variables that are local to the currently qualified routine.



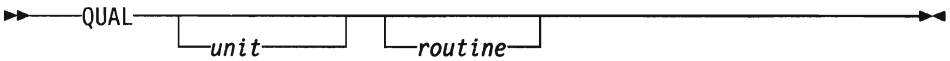
Minimum abbreviation: L

QUAL Command

QUAL sets the scope of variables to be displayed.

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

If you have two or more procedures with the same name in a module, the routine qualified may be unpredictable.



Where	Specifies
QUAL	The command keyword. *Minimum abbreviation*: Q
unit	The name of a VS Pascal unit.
routine	The name of a procedure or function in the unit.

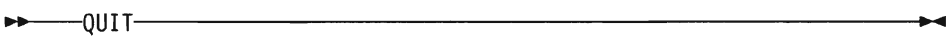
If you do not specify a unit and/or a routine name, the defaults are taken from the current qualification. The defaults are applied as follows:

- The unit name defaults to the current qualification.
- The routine defaults to the main program if the associated unit is a program unit.

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.

QUIT Command

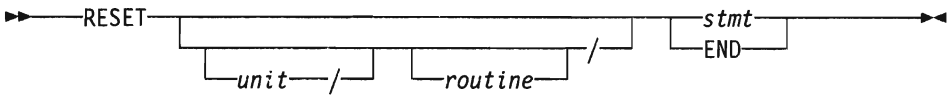
QUIT terminates the program immediately. This command is synonymous with END.



Minimum abbreviation: QUIT

RESET Command

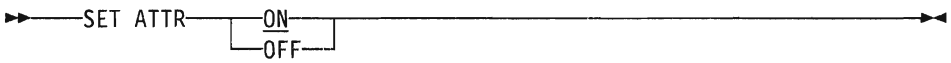
RESET removes a specified breakpoint. The defaults are the same as the BREAK command.



Where	Specifies
RESET	The command keyword. *Minimum abbreviation*: R
unit	The name of a VS Pascal unit.
routine	The name of a procedure or function in the unit.
stmt	The number of a statement in the designated routine.
END	The breakpoint is in the epilog of the routine before the routine's return.

SET ATTR Command

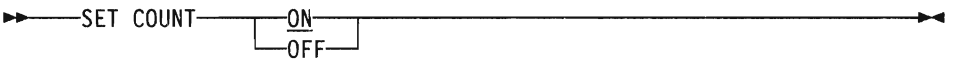
SET ATTR sets the default way in which variables are viewed. The default may be overridden on the variable viewing command.



Where	Specifies
SET ATTR	The command keyword. *Minimum abbreviation*: S A
ON	Variable attribute information will be displayed by default. The attributes are the data type, storage class, length if relevant, and the routine where the variable was declared.
OFF	Variable attribute information will not be displayed by default.

SET COUNT Command

SET COUNT initiates and terminates 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 or when a DISPLAY COUNTS command is issued.



Where	Specifies
SET COUNT	The command keyword. *Minimum abbreviation*: S C
ON	Statement counting is on.
OFF	Statement counting is off.

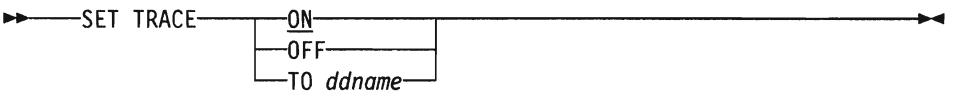
Notes:

1. Statement counting may also be initiated with the COUNT run-time option.
2. Counting is done on a "basic block" basis. A basic block is a group of statements entered only from the first statement. This implies that a line can be counted as executed if any line in the basic block has been executed.

SET TRACE Command

SET TRACE activates or deactivates program tracing. Program tracing provides the user with a list of every statement executed in the program. This is useful for following the execution flow.

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.

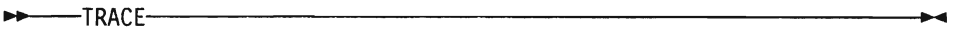


Where	Specifies
SET TRACE	The command keyword. *Minimum abbreviation*: S T
ON	Program tracing is on.
OFF	Program tracing is off.
TO *ddname*	Trace output will be sent to *ddname*.

Note: Tracing is done on a "basic block" basis. A basic block is a group of statements entered only from the first statement. This implies that a line can be counted as executed if any line in the basic block has been executed.

TRACE Command

TRACE produces a routine trace-back. The routines on the current invocation chain are listed along with the most recently executed statement in each.

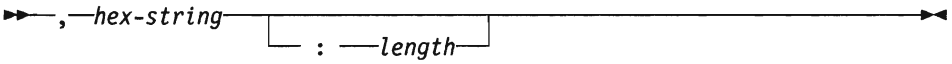


Minimum abbreviation: T

For more information regarding trace-backs, see "Reading a VS Pascal Trace-Back Report" on page 73.

Viewing Storage

,hex-string displays the contents of a specific storage location. The dump is in both hexadecimal and character formats.



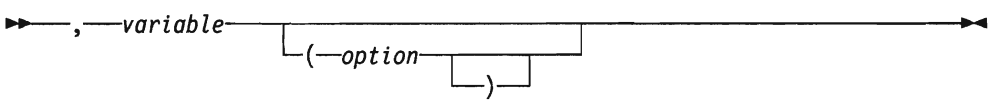
Where	Specifies
hex-string	The beginning storage location to be dumped. It must be a hexadecimal number surrounded by single quotes and followed by an 'X' (for example '35D05'X).
length	The number of bytes to be dumped. It must be a decimal number. If *length* is not specified, storage is dumped for 16 bytes.

Examples

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

Viewing Variables

,variable displays the contents of a variable during program execution.



Where	Specifies
variable	A VS Pascal variable.
option	Either ATTR or NOATTR. The default is taken from the SET ATTR command. The initial default is NOATTR. If ATTR is specified, attributes of the variable are displayed along with the value of the variable. The attributes are the data type, storage class, length if relevant, and the routine where the variable was declared.

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, you are informed that the name is not found. If the name resolves to an automatic variable for which no activation currently exists, you are 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. You may view an array slice by specifying fewer indices than the declared dimension of the array. The missing indices must be the rightmost ones.

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]
```

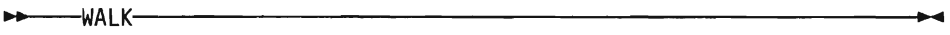
If the variable being viewed has not been assigned a value, then the results depend on the variable's type:

- If the variable is a scalar, pointer, set or string type, the word *uninitialized* will be written.
- If the variable is of a structured type (array, record, and so forth), then the contents will be printed in hexadecimal; each byte of the the variable which is uninitialized will have the hexadecimal value X'FE'.

WALK Command

WALK causes the program to either start executing or resume executing. The program execution will continue for exactly one statement in the current routine and then you will be returned to the debugging environment. This implies that if the statement is a routine call, the routine will be executed as one statement. However, if a breakpoint was set in the routine, the routine would stop at the breakpoint (if appropriate).

This command is useful for single stepping through a section of code.



Minimum abbreviation: W

Chapter 16. VS Pascal Register and Storage Usage

This chapter discusses how VS Pascal uses registers and storage.

Linkage Conventions

VS Pascal uses standard System/370 linkage conventions with several additional restrictions. The result is that VS Pascal may call any program that requires standard conventions and may be called by any program that adheres to the additional VS Pascal restrictions.

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

- Register 1 — points to the parameter list
- Register 11 — points to the dynamic storage area (DSA) of the main program
- Register 12 — points to the VS Pascal 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.

VS Pascal requires that the parameter register (register 1) be pointing into the dynamic storage area (DSA) stack in such a way that 144 bytes before the register 1 address is an available save area.

Register Usage

The list below describes how each general register is used within a VS Pascal program. The floating-point registers are used for computation on REAL data types.

- Registers 0 and 1 — temporary work registers for the compiler and standard linkage usage on calls
- Registers 3, 4, 5, 6, 7, 8, and 9 — Registers assigned by the compiler for computation and for data base registers
- Registers 2 and 10 — Code base registers of the currently executing routine
- Register 11 — Address of the DSA of the main program
- Register 12 — Always points to VS Pascal communication work area
- Register 13 — Always points to the local DSA
- Registers 14 and 15 — Temporary work registers for the compiler and standard linkage usage on calls.

Routine Invocation

Each invocation of a VS Pascal routine must acquire a *dynamic storage area* (DSA). This storage is allocated and deallocated in a LIFO (last in/first out) stack. If the stack is filled, 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 VS Pascal 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 prolog 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 97 illustrates the condition of the stack and relevant registers immediately at the start of a routine.

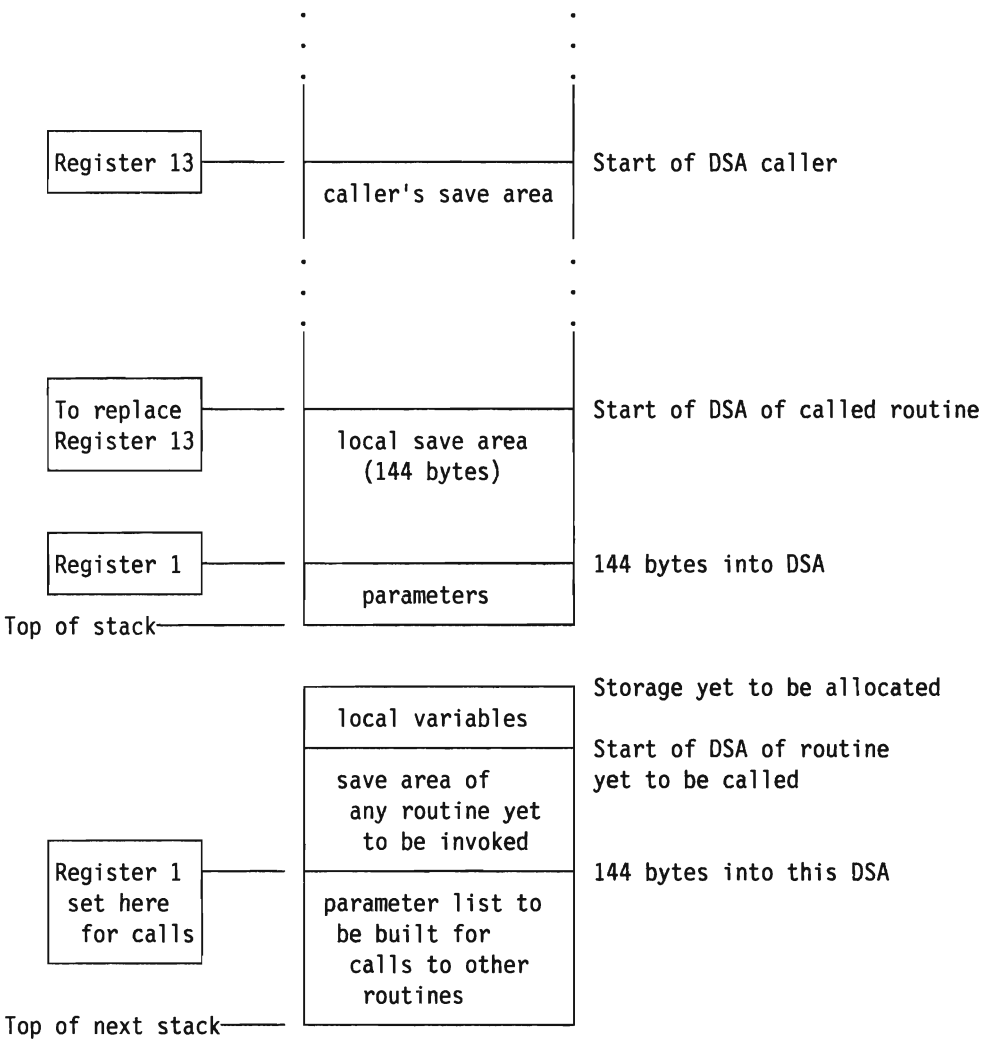


Figure 97. Snapshot of Stack and Relevant Registers at Start of Routine

Procedure and Function Format

Every VS Pascal procedure or function is arranged in the order shown in Figure 98. 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.

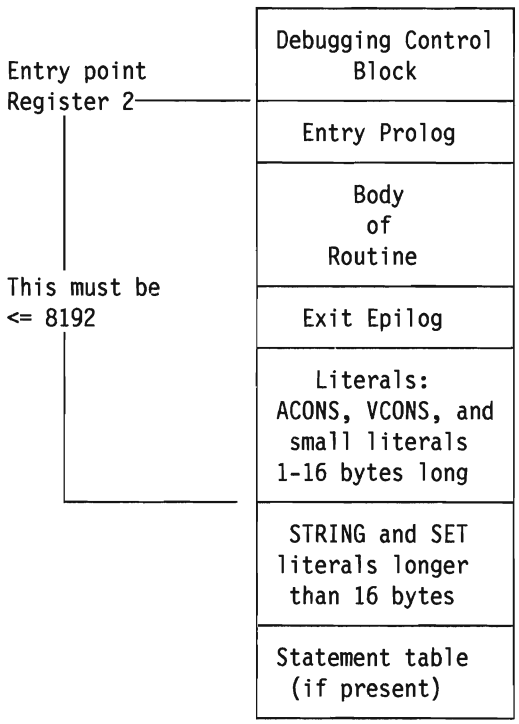


Figure 98. Routine Format

Storage Mapping

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

Storage for Automatic Variables

Variables declared locally to a routine through the VAR declaration 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.

Storage for Static Variables

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 through the STATIC declaration, regardless of its scope, is assigned a unique offset within this control section.

Storage for DEF Variables

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, which may be used to communicate with FORTRAN subroutines. The name of the section is derived from the first eight characters of the variable's name.

Storage for Dynamic Variables

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

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.

Data Size and Boundary Alignment of the VS Pascal Data Types

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

The Predefined Data Types

Figure 99 shows the storage requirements and boundary alignment of variables declared with a predefined type.

Data Type	Size in Bytes	Boundary Alignment
ALFA	8	Byte
ALPHA	16	Byte
BOOLEAN	1	Byte
CHAR	1	Byte
GCHAR	2	Halfword
GSTRING(*length*)	*length**2+2	Halfword
INTEGER	4	Fullword
SHORTREAL	4	Fullword
REAL	8	Doubleword
STRING(*length*)	*length*+2	Halfword
STRINGPTR	8	Fullword
TEXT	64	Fullword

Figure 99. Storage Mapping of Data for Predefined Types

Enumerated Scalar Data Types

An enumerated scalar variable with 256 or fewer possible distinct values will occupy 1 byte and will be aligned on a byte boundary. If the scalar defines more than 256 values, then it will occupy a halfword and will be aligned on a halfword boundary.

Subrange Data Types

A subrange scalar data type 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 data type is mapped as indicated in the table of Figure 100. 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	Halfword
0..65535	2	Halfword
0..16777215	3	Byte
-8388608..8388607	3	Byte
OTHERWISE	4	Fullword

Figure 100. Storage Mapping of Subrange Scalars

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

RECORD Data Types

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 fullword because its field A1 requires a fullword 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 (*fullword 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 101. Alignment of Records

Packed records are always aligned on a byte boundary.

ARRAY Data Types

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
```

FILE Data Types

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

SET Data Types

SET data types 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 nonsubrange 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 a 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 nonsubrange 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 data 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 24 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 102 indicates how the set will be mapped in storage.

Range of M	Size in Bytes	Alignment
1 <= M <= 8	1	Byte
9 <= M <= 16	2	Halfword
17 <= M <= 24	3	Byte
25 <= M <= 32	4	Fullword
33 <= M <= 256	(M+7) DIV 8	Byte

Figure 102. Storage Mapping of SET Data Types

SPACE Data Types

A variable declared as 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;
```

Chapter 17. VS Pascal Parameter Passing

VS Pascal passes parameters in several different ways depending on how you declared the parameter. 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.

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.

Routine Heading

```
PROCEDURE PROC(VAR I:INTEGER);
```

Routine Invocation

```
PROC(J);
```

Parameter List
Address of J.

If the variable is a structure containing a file, an integer file key is also passed to the routine by value.

Routine Heading

```
PROCEDURE PROC(VAR A:FILEARRAY);
```

Routine Invocation

```
PROC(A);
```

Parameter List
Address of A and value of file key.

If the parameter is a conformant string, the maximum length of the string is passed to the routine by value.

Routine Heading

```
PROCEDURE PROC( CONST S: STRING);
```

Declaration of the Actual Parameter

```
VAR
  ST : STRING(64);
```

Routine Invocation

```
PROC(ST);
```

Parameter List
The address of ST and 64.

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. For actual parameters the same size as the formal parameter and not packed, the parameter list contains the address of the actual parameter. Otherwise, the parameter list contains the address of a temporary variable containing the value of the actual parameter.

Routine Heading

```
PROCEDURE PROC(CONST I: INTEGER);
```

Routine Invocation

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

Parameter List

Address of a storage location that contains the value of J+5. For actual parameters which are variables of the same size as the formal parameter and not packed, the address passed is the address of the variable. Otherwise, the address is of a temporary containing the value of the actual parameter.

If the parameter is a conformant string, only the address of the string is passed to the routine.

Routine Heading

```
PROCEDURE PROC( VAR S: STRING);
```

Declaration of the Actual Parameter

```
VAR
  ST : STRING(64);
```

Routine Invocation

```
PROC(ST);
```

Parameter List

The address of ST.

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 preinitialized 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 and address of *alpha*.

Passing Procedure or Function Parameters

For procedures or functions which are passed as parameters, the address of the routine and six additional words reserved for VS Pascal use are placed in the parameter list.

Routine Heading

```
PROCEDURE PROC(
  FUNCTION X(Y: REAL): REAL );
```

Routine Invocation

```
PROC(COS);
```

Parameter List

Address of the COS routine and six additional words.

Function Results

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

Routine Heading

```
FUNCTION FUNC(C: CHAR):INTEGER;
```

Routine Invocation

```
I := FUNC('L');
```

Parameter List

Address of returned integer result and value of character *L*.

FORTRAN Routines

Routines using the same parameter passing conventions as FORTRAN pass only VAR or CONST parameters, and the high-order bit in the last parameter in the set.

If the routine is declared as a function and is not floating-point, the result is returned in register 0. If the routine is declared as a function and is floating-point, the result is returned in floating-point register 0.

Note: There is no double parameter list for string arguments.

Routine Heading

```
FUNCTION FUNC(CONST I: INTEGER):INTEGER;FORTRAN;
```

Routine Invocation

```
I := FUNC(J + 5);
```

Parameter List

Address of temporary with the high-order bit on containing the value of J+5.

GENERIC Procedures

GENERIC procedures have no formal parameters. Their parameters are "declared" when they are called. The last actual parameter specified on a call to a GENERIC routine has its high-order bit turned on.

Routine Heading

```
PROCEDURE P; GENERIC;
```

Routine Invocation

```
P(VAR I, CONST J+5);
```

Parameter List

Address of I and address of a temporary with the high-order bit on containing the value of J+5.

Chapter 18. Managing Storage

This section explains how you manage storage in VS Pascal. It also explains how to use storage efficiently.

Storage Management

VS Pascal provides three levels of storage management. You allocate and deallocate storage for:

- Individual dynamic variables
- Groups of variables (subheaps or marks)
- Heaps (groups of subheaps and dynamic variables).

In a multiple-module application, multiple storage heaps provide a simple method for isolating the storage used by different program components. By isolating the storage, you can often reduce storage fragmentation.

Dynamic Variables

Storage is allocated for a dynamic variable using the NEW routine and deallocated with the DISPOSE routine. The amount of storage allocated depends on the data type of the variable.

Figure 103 gives an example of managing storage at the individual variable level.

```
PROGRAM DYN_VAR;

TYPE
  SMALL_RECORD = RECORD
                   AT_BAT: INTEGER;
                   HITS:   INTEGER;
                 END;

VAR
  ALICIA, BOBBY, CHARLIE: @SMALL_RECORD;

BEGIN
  NEW(ALICIA);  (* Allocate a SMALL_RECORD and place its      *)
                (* address in the variable Alicia.           *)
  NEW(BOBBY);   (* Allocate a SMALL_RECORD and place its      *)
                (* address in the variable Bobby.            *)
  NEW(CHARLIE); (* Allocate a SMALL_RECORD and place its      *)
                (* address in the variable Charlie.          *)

  (* Here we might do a lot of calculations that decide we    *)
  (* no longer need the storage associated with Bobby or      *)
  (* or Charlie.                                              *)

  DISPOSE(BOBBY);  (* Deallocate the SMALL_RECORD associated  *)
                   (* with the variable Bobby.               *)
  DISPOSE(CHARLIE);(* Deallocate the SMALL_RECORD associated  *)
                   (* with the variable Charlie.             *)
END.
```

Figure 103. Managing Individual Dynamic Variables Using NEW and DISPOSE

Subheaps

Subheaps are used to define logical groups of dynamic variables. When you call the MARK routine, you define the beginning of a subheap. After a call to MARK, all dynamic variables allocated in that heap become members of the subheap. The MARK routine simplifies the freeing of storage used for the dynamic variables by defining a logical group (the subheap). A subheap can be freed with one call to the RELEASE routine, instead of requiring a separate DISPOSE call for each dynamic variable.

Figure 103 on page 199 can be made simpler and faster by doing the following:

```
PROGRAM SUB_HEAPS;

TYPE
  SMALL_RECORD = RECORD
                   AT_BAT: INTEGER;
                   HITS:   INTEGER;
                 END;
  DUMMY = @INTEGER;  (* This type will be used for subheap    *)
                     (* identifiers.                          *)

VAR
  ALICIA, BOBBY, CHARLIE: @SMALL_RECORD;
  SUB_HEAP: DUMMY;

BEGIN
  NEW(ALICIA);    (* Allocate a SMALL_RECORD and place its     *)
                  (* address in the variable Alicia.           *)
  MARK(SUB_HEAP);(* Define the beginning of a subheap.        *)
  NEW(BOBBY);     (* Allocate a SMALL_RECORD and place its     *)
                  (* address in the variable Bobby.            *)
  NEW(CHARLIE);   (* Allocate a SMALL_RECORD and place its     *)
                  (* address in the variable Charlie.          *)

  (* Here we might do a lot of calculations that decide we     *)
  (* no longer need the storage associated with Bobby or       *)
  (* Charlie.                                                  *)

  RELEASE(SUB_HEAP);  (* Deallocate ALL dynamic variables     *)
                      (* allocated since the call to          *)
                      (* MARK(SUB_HEAP).                      *)
END.
```

Figure 104. Managing a Subheap Using MARK and RELEASE

Heaps

The most general storage entity in VS Pascal is a heap, which contains dynamic variables and subheaps. A heap is an extendable block of storage acquired by the VS Pascal environment for use by Pascal applications.

The idea of the current (or active) heap is central to the heap concept. In a multiple heap application, it is important to know the current heap because NEW and MARK operate on the current heap. However, DISPOSE and RELEASE free storage in any heap, regardless of whether or not it is the current heap.

Allocating a separate HEAP for each data structure allows the entire structure to be freed (using DISPOSE) even if the structure was not allocated in a stack-line manner that RELEASE can use.

The default size of a new heap is 12 kilobytes, and the amount it is extended upon overflow is also 12 kilobytes. At run time, you may change these values by using either the HEAP option (see "HEAP Option" on page 169) or with the *init* and *incr* parameters of the NEWHEAP routine (refer to the *VS Pascal Language Reference* for more information). A program can also control other attributes, such as the location of heaps in relation to the 16-megabyte addressing line.

Heaps are created with the NEWHEAP routine, specified as the current (active) heap with the USEHEAP routine, and disposed of with the DISPOSEHEAP routine. To identify the current heap use the QUERYHEAP routine.

Figure 105 shows how heaps are used within a module for storage management.

```
PROGRAM HEAPS;

TYPE
  SMALL_RECORD = RECORD
          AT_BAT: INTEGER;
          HITS:   INTEGER;
         END;
  HEAPPTR = @INTEGER;  (* This type will be used for subheap   *)
                       (* and heap identifiers                  *)

VAR
  ALICIA, BOBBY, CHARLIE, DIANE, EDWARD, FRANCES: @SMALL_RECORD;
  HEAP_ONE, HEAP_TWO: HEAPPTR;
  SUB_HEAP_ONE, SUB_HEAP_TWO: HEAPPTR;

BEGIN

  NEWHEAP(HEAP_ONE);  (* Create a heap and place its          *)
                      (* identifier in the variable HEAP_ONE. *)
  NEWHEAP(HEAP_TWO);  (* Create a heap and place its          *)
                      (* identifier in the variable HEAP_TWO. *)
  USEHEAP(HEAP_ONE);  (* Make HEAP_ONE the current heap.      *)

  NEW(ALICIA);    (* Allocate a SMALL_RECORD in the heap       *)
                  (* HEAP_ONE and place its address in the     *)
                  (* variable Alicia.                          *)

  MARK(SUB_HEAP_ONE); (* Define the beginning of a subheap of *)
                      (* HEAP_ONE and place its identifier in *)
                      (* the variable SUB_HEAP_One.           *)

  NEW(BOBBY);     (* Allocate a SMALL_RECORD in the heap       *)
                  (* HEAP_ONE and place its address in the     *)
                  (* variable Bobby.                           *)
  NEW(CHARLIE);   (* Allocate a SMALL_RECORD in the heap       *)
                  (* HEAP_ONE and place its address in the     *)
                  (* variable Charlie.                         *)
```

Figure 105 (Part 1 of 2). Managing Storage Using Two Heaps

```
  USEHEAP(HEAP_TWO);  (* Make HEAP_TWO the current heap.      *)

  NEW(DIANE);     (* Allocate a SMALL_RECORD in the heap      *)
                  (* HEAP_TWO and place its address in the    *)
                  (* variable Diane.                          *)

  MARK(SUB_HEAP_TWO); (* Define the beginning of a subheap of *)
                      (* HEAP_TWO and place its identifier in *)
                      (* the variable SUB_HEAP_Two.           *)

  NEW(EDWARD);    (* Allocate a SMALL_RECORD in the heap      *)
                  (* HEAP_TWO and place its address in the    *)
                  (* variable Edward.                         *)
  NEW(FRANCES);   (* Allocate a SMALL_RECORD in the heap      *)
                  (* HEAP_TWO and place its address in the    *)
                  (* variable Frances.                        *)

  (* See the table below for the results of some of the      *)
  (* calls that may be made at this point.                   *)

END.
```

Figure 105 (Part 2 of 2). Managing Storage Using Two Heaps

The following table explains the effects of some other calls that you can make at the end of the program in Figure 105 on page 201.

Call	Result
DISPOSE(EDWARD);	Deallocates the dynamic variable whose address is stored in EDWARD.
DISPOSE(ALICIA);	Deallocates the dynamic variable whose address is stored in ALICIA, even though it is not part of the current heap.
RELEASE(SUB_HEAP_ONE);	Deallocates all dynamic variables allocated in HEAP_ONE since the creation of the subheap SUB_HEAP_ONE; that is, BOBBY and CHARLIE. Additionally, the subheap identifier SUB_HEAP_ONE is deallocated.
RELEASE(SUB_HEAP_TWO);	Deallocates all dynamic variables allocated in HEAP_TWO since the creation of the subheap SUB_HEAP_TWO; that is, EDWARD and FRANCES. Additionally, the subheap identifier SUB_HEAP_TWO is deallocated.
DISPOSEHEAP(HEAP_ONE);	Deallocates the entire heap whose pointer is stored in HEAP_ONE. This deallocates the dynamic variables ALICIA, BOBBY, and CHARLIE, and the subheap identifier SUB_HEAP_ONE.
DISPOSEHEAP(HEAP_TWO);	Deallocates the entire heap whose pointer is stored in HEAP_TWO. This deallocates the dynamic variables DIANE, EDWARD, and FRANCES, and the subheap identifier SUB_HEAP_TWO. Additionally, note that because HEAP_TWO is the current heap, there will be no current heap following the call.

Using Storage Intelligently

Imagine an application containing three separate modules, each using dynamic variables and subheaps. Upon invocation, each module would:

- Call QUERYHEAP and save the identifier of the current heap
- Call NEWHEAP to create a heap for its own use
- Call USEHEAP to make that heap the current heap
- Perform its designated task
- Call USEHEAP to restore the current heap to its invocation value
- Call DISPOSEHEAP to free its heap.

If each module in the application follows this practice, overall performance may be improved because "well behaved" storage algorithms are isolated from the fragmentation caused by those less well behaved. Additionally, a given module's storage can be tuned to a specific size. Note, however, that creating a heap usually involves requesting storage from the operating system, a relatively expensive process in terms of application performance. Calls to NEWHEAP should be coded with this in mind.

Figure 106 is an example of "well behaved" storage algorithms.

```
SEGMENT APPL_PART;

TYPE
  HEAPPTR @INTEGER;  (* This type will be used for subheap       *)
                     (* and heap identifiers.                    *)
PROCEDURE APPL_PART; EXTERNAL;
PROCEDURE APPL_PART;

VAR
  SAVED_HEAP, MY_HEAP: HEAPPTR;

BEGIN
  QUERYHEAP(SAVED_HEAP);     (* Save the calling module's heap.  *)
  NEWHEAP(MY_HEAP);          (* Create a heap for use by this    *)
                             (* module.                          *)
  USEHEAP(MY_HEAP);          (* Make it the current heap.        *)
     .
     .

  (* The application code would go here.                         *)
     .
     .

  USEHEAP(SAVED_HEAP);       (* Restore the current heap to the  *)
                             (* caller's heap.                   *)
  DISPOSEHEAP(MY_HEAP);      (* Deallocate all storage used by   *)
                             (* this module.                     *)
END;
```

Figure 106. Saving and Restoring the Current Heap

See the *VS Pascal Language Reference* for more information on the NEW, DISPOSE, MARK, RELEASE, NEWHEAP, DISPOSEHEAP, QUERYHEAP, and USEHEAP routines.

Chapter 19. Performance Considerations

This chapter describes the optimizations VS Pascal performs, and discusses what you can do to make your program more efficient.

Optimizations Performed by VS Pascal

This section describes the various optimizations performed by VS Pascal. All optimizations listed are performed automatically by VS Pascal, with the exception of common subexpression elimination, which is controlled by the OPTIMIZE compile-time option.

Constant Folding

Constant folding refers to the process of replacing an expression consisting of constants with a single constant.

```
A := 3 * SQR(20);
```

would be logically replaced with

```
A := 1200;
```

While all operators are eligible for constant folding, only the functions that can be used in constant expressions are eligible for constant folding. Therefore, an expression such as SUBSTR('ABC',1,2) would still involve a library call.

In-Line Code for Predefined Routines

Certain predefined routines are implemented using in-line code instead of calls to library routines. These routines include:

ABS	LBOUND	PRED
ADDR	LENGTH	ROUND
CHR	LOWEST	SIZEOF
EOF	MAX	SQR
EOLN	MAXLENGTH	STR
FLOAT	MIN	SUCC
GSTR	ODD	TRUNC
HBOUND	ORD	
HIGHEST	Ordinal conversion	

Expression Simplification

Certain expressions will be reordered to be more efficient. Boolean expressions containing the "NOT" operator are transformed so that the "NOT" is removed. For example, the statement

```
IF NOT (I > J) THEN
```

is logically transformed to

```
IF I <= J THEN
```

Boolean Short-Circuiting

Boolean short-circuiting (also known as "anchor pointing" and "partial evaluation") allows Boolean expressions containing the "AND" and "OR" operators to only have the left side expression evaluated if that is enough to resolve the condition. For example, given Boolean expressions "A" and "B", the expression

```
IF A AND B THEN statement;
```

will be transformed to

```
IF A THEN IF B THEN statement;
```

The expression

```
IF A OR B THEN statement;
```

will be transformed to

```
  IF A THEN GOTO LAB1
  ELSE IF ¬B THEN  GOTO LAB2;
LAB1:
  statement;
LAB2:
```

As a less trivial case consider

```
IF ((I < 0) AND (J < 0))
          OR
   ((I > 9) AND (J > 9))
THEN
  I := J;
```

This statement would be transformed to evaluate as follows:

```
  IF NOT (I < 0) THEN GOTO LAB2
  IF J < 0 THEN GOTO LAB3
LAB2:
  IF NOT (I > 9) THEN GOTO LAB1
  IF NOT (J > 9) THEN GOTO LAB1
LAB3:
  I := J;
LAB1:
```

Boolean expressions which are being assigned to variables are transformed to conditional expressions which produce a value of TRUE or FALSE within a temporary. For example:

```
B := (I > 0) OR (I = J);
```

is transformed to

```
  IF I > 0 THEN GOTO LAB1
  ELSE IF I <> J THEN GOTO LAB2;
LAB1:
  B := TRUE;
  GOTO LAB3;
LAB2:
  B := FALSE;
LAB3:
```

Note: Boolean short-circuiting can cause problems if the right operand of the Boolean operator is a function call and subsequent statements in the program require that the function was called (in other words, the program relies on a side-effect of the function). For example, the following program would print "PASS" if short-circuiting is not done, and "FAIL" if short-circuiting is done.

```
PROGRAM P;
VAR
  B1, B2 : BOOLEAN;
FUNCTION F(VAR B : BOOLEAN) : BOOLEAN;
BEGIN
  F := B;
  B := NOT B;
END;
BEGIN
  B1 := TRUE;
  B2 := TRUE;
  IF B1 OR F(B2) THEN
    IF B2 THEN
      WRITELN('FAIL')
    ELSE WRITELN('PASS')
  ELSE WRITELN('FAIL');
END.
```

Cascaded Branches

A cascaded branch is when the target of one branch is an unconditional branch. The first branch can be optimized so that its target is the target of the unconditional branch.

The following program helps illustrate this.

```
PROGRAM BRANCH;
LABEL 1;
VAR
  I : INTEGER;
BEGIN
  I := 3;
  CASE I OF
    MININT..-1 :
      WRITELN('FAIL CASE');
    0:
      WRITELN('FAIL CASE');
    1,2:
      WRITELN('FAIL CASE');
    3:
      GOTO 1;
    4..MAXINT:
      WRITELN('FAIL CASE');
  END;
  WRITELN('FAIL TEMP');
1:
  WRITELN('PASS TEMP');
END.
```

The CASE statement will generate a branch to the label indicated by the selector, which in this case is an unconditional branch (GOTO) to label 1. This will be optimized so that when the CASE selector is 3, control will transfer to label 1 directly instead of to the GOTO.

Partial Dead Code Elimination

Code which immediately follows an unconditional branch and is not the target of another branch is unreachable. This "dead code" can be deleted.

The example in "Cascaded Branches" on page 207 illustrates this. Normally, each case in a CASE statement ends with an unconditional branch past the end of the CASE statement. Because the statement for case 3 is itself an unconditional branch (GOTO), this last branch could be removed. In fact, because the branch to the GOTO was a cascaded branch, all of case 3 is dead code, so case 3 would result in no machine instructions being generated.

Set Operations

The set relational operators for subset and superset are optimized for better code-generation. Relational expressions of the form

```
S1 - S2
```

where "S1" and "S2" are set expressions, are transformed to

```
(S1 * ~S2) = S1
```

The latter form is more suitable for machine code generation. Similarly, expressions of the form

```
S1 >= S2
```

are transformed to

```
(S2 * S1) = S2
```

The set difference operator is optimized for better code-generation. Expressions of the form

```
S1 - S2;
```

are transformed to

```
S1 * ¬S2;
```

Strength Reduction

Multiplication and division by integer powers of 2 are transformed to arithmetic shifts. For example,

```
N * 64
```

is transformed to

```
N >> 6
```

and

```
N DIV 32
```

is transformed to

```
N << 5
```

Modulo operations in which the right side operand is an integer power of 2 and the left side operand is non-negative are transformed to masking operations. For example, assuming N is unsigned,

```
N MOD 256
```

is transformed to

```
N AND 255
```

Array References

Subscripted addresses are transformed for easier translation to efficient machine code.

Given a subscripted variable reference of the form "A[N]", where A is of the type "ARRAY[I..J] of T", the associated address of A[N] can be computed as

```
ADDR(A) + (N-I)*SIZEOF(T)
```

where "ADDR(A)" is the address of array A, and "SIZEOF(T)" is the length of each element of the array.

The subtraction in the above expression can be eliminated by "origining" the array as follows:

```
(ADDR(A)-I*SIZEOF(T)) + N*SIZEOF(T)
```

in which the first operand of the addition is computed at compile-time.

Constant subscripts, such as A[3], have their addresses completely computed at compile-time.

Unnesting of Function Calls

Actual parameters of routine calls which are themselves function calls can often cause difficulty in code generation. If all routines calls are "unnested" so that no call will take place while the parameter list of another call is being constructed, this problem is avoided. This optimization simplifies managing the run-time stack.

For example, the statement

```
X := SIN(ARCTAN(Y));
```

would be evaluated as

```
temp := ARCTAN(Y);
X := SIN(temp);
```

where "temp" is a compiler generated temporary.

Common Subexpression Elimination

Common subexpression elimination involves evaluating an expression once and saving its value instead of computing the same expression several times. For example,

```
I := W + Y * Z DIV 3;
J := X + Y * Z DIV 3;
```

would be logically replaced with

```
temp := Y * Z DIV 3;
I := W + temp;
J := X + temp;
```

where "temp" would be a register.

This optimization is performed over a range of instructions known as a "basic block". A basic block is a group of statements that is only entered from the first statement. For example, common subexpression would not be performed in the following case because the assignments are in different basic blocks.

```
IF B THEN
  I := W + Y * Z DIV 3
ELSE
  I := X + Y * Z DIV 3;
```

Figure 107 shows the effects of common subexpression elimination

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 107. Example of the Differences Between Optimized and Unoptimized Code

Note: Because some values are kept in registers at certain times, attempting to assign values to variables in an ONERROR procedure may appear to fail if the program is compiled with the OPTIMIZE compile-time option in effect.

In the following program, if file F does not exist and the OPTIMIZE option is in effect, the program will print out "File opened OK" even if the file did not open successfully. This is because the variable FILE_ERROR was kept in a register, and though the storage copy was updated in ONERROR, the IF test used the value in the register.

```
PROGRAM ERROR;
VAR
  FILE_ERROR : INTEGER;
  F : TEXT;
%INCLUDE ONERROR;
PROCEDURE ONERROR;
BEGIN
  FACTION := [];
  FILE_ERROR := FERROR;
END;
BEGIN
  FILE_ERROR := 0;
  RESET(F);
  IF FILE_ERROR <> 0 THEN WRITELN('File error ',FILE_ERROR:1)
  ELSE WRITELN('File opened OK');
END.
```

Memory References

Many target machines have limits on the range of memory that can be directly addressed from a base register. The IBM System/370, for example, has a limit of 0 to 4095 bytes. When an address has a displacement which is outside this range, the address must be explicitly computed.

One of the transformations performed by the compiler is to expand memory references which are not directly addressable into explicit address computations. This permits the common subexpression eliminator to optimize such computations.

As an example, consider the following procedure.

```
PROCEDURE MEMREF;
VAR
  HUGE : ARRAY[1..10000] OF CHAR;
  A : INTEGER;
  B : INTEGER;
BEGIN
  B := 3;
  A := B;
END;
```

With OPTIMIZE on, the following code is generated.

```
*  B := 3;
         LR    03,13
         AH    03,=H'8192'
         LA    04,3
         ST    04,1956(,03)
*  A := B;
         ST    04,1952(,03)
```

Without the optimization, code similar to the following would be generated, taking more code.

```
*  B := 3;
         LR    03,13
         AH    03,=H'10148'
         LA    04,3
         ST    04,0(,03)
*  A := B;
         LR    03,13
         AH    03,=H'10144'
         ST    04,0(,03)
```

If additional references to variables past HUGE were made, either more code would be needed (reloading register 13 into register 3 for each reference), or an additional register would be needed (to save register 13).

Range Checking

Transformations are performed on subrange checks involving expressions of the forms:

```
E + C
E - C
E * C
E DIV C
```

where "E" is an integer expression, and "C" is an integer constant. The transformation moves the check down to the expression "e" as illustrated in the following example.

```
VAR
  N : 1..10;
  M : INTEGER;
BEGIN
  .
  .
  N := M - 15;
  .
  .
END;
```

In the first compiler pass, code will be generated for the expression "M-15" to verify that it is within the range 0 to 10. The second compiler pass will rewrite the checking code so that variable "M" will be checked instead of the whole expression. In this case, "M" must lie in the range 16 to 25.

The purpose of this transformation is to help the common subexpression eliminator (in the third pass) remove redundant range checks. Also, subrange variables whose lower bound has an ordinal value of zero will not have low-bound checking code generated; an unsigned check for the high-bound will be generated instead.

Note: Due to these optimizations, some checks normally resulting in low-bound checking errors will actually result in high-bound checking errors and vice versa.

Making Your Programs More Efficient

Variable Declaration

VS Pascal allocates automatic variables in the order they are declared. Due to System/370 architecture, accessing variables in the first 4K of a DSA is more efficient than accessing variables allocated after the first 4K. Therefore, small and frequently accessed variables should be declared first in a routine.

Array Bounds

Large lower bounds on arrays can cause array origining to be omitted, requiring additional calculations at run-time. The following program illustrates this.

```
PROGRAM ARRBOUND;
CONST
  ALB = 1;
  AHB = 10;
  BLB = 20000001;
  BHB = 20000010;
VAR
  A : ARRAY[ALB..AHB] OF INTEGER;
  B : ARRAY[BLB..BHB] OF INTEGER;
  I : INTEGER;
BEGIN
  IF SIZEOF(A) <> SIZEOF(B) THEN WRITELN('FAIL ARRBOUND');
  FOR I := ALB TO AHB DO
    A[I] := I;
  FOR I := BLB TO BHB DO
    B[I] := I;
END.
```

The code generated for each array element assignment is shown below.

```
*    A[I] := I;
         L     03,224(,13)
         LR    04,03
         SLA   04,2
         ST    03,140(04,13)
*    B[I] := I;
         LR    03,13
         A     03,=F'-80003072'
         L     04,224(,13)
         LR    05,04
         SLA   05,2
         ST    04,3252(05,03)
```

Record Field Accessing

Using a WITH statement on a complicated field reference that is used multiple times can sometimes speed up your program. This is because computing the address to the field is only done once and saved in memory. However, because the address of the field is stored in memory when using a WITH statement, there are cases where using a WITH statement will be less efficient than explicit field references, especially if common subexpression elimination can be used.

Program Parameters

Placing INPUT in the program parameter list causes a RESET(INPUT) to be performed; therefore, if INPUT is not used in your program, delete it from the program parameter list.

A similar optimization is possible for OUTPUT because placing OUTPUT in the program parameter list causes a REWRITE(OUTPUT) to be performed.

File Closing

Calling the CLOSE procedure is never needed except under those conditions mentioned in "Closing a File" on page 70, or if you intentionally associate different Pascal file variables with the same external file.

You may wish to call CLOSE to make it obvious that a file is closed, or to guarantee that all data is written to a file in case a severe error occurs in your program.

Use of Value and Constant Parameters

When a structured variable is to be passed to a routine where it will not be modified, using the CONST parameter passing mechanism will be more efficient than using pass-by-value.

Note: When using CONST parameters, you must be careful that you do not threaten an "alias" of the CONST parameter name. An alias is defined as the name of the actual parameter or the name of a VAR parameter which represents the same value, or any alias of those names. The following example illustrates this.

```
PROGRAM ALIAS(OUTPUT);
TYPE
  ARR = ARRAY[1..20] OF INTEGER;
VAR
  A1, A2, A3 : ARR;
PROCEDURE P(VAR B1 : ARR; CONST B2, B3 : ARR);
BEGIN
  B1[1] := 3;
  if B2[1] <> A3[1] THEN WRITELN('B2 modified&ssq];
  A2[1] := 3;
  if B3[1] <> A3[1] THEN WRITELN('B3 modified&ssq];
END;
BEGIN
  A1 := ARR(1234:20);
  A2 := A1;
  A3 := A1;
  P(A1, A1, A2);
END.
```

VALUE Initializations

It is more efficient to use a VALUE declaration to initialize a STATIC or DEF variable than to use an assignment statement at the beginning of a routine. This is because the linkage editor performs this initialization if a VALUE declaration is used.

Note: If a routine modifies a STATIC or DEF variable, the next time the routine is called the variable will have the new value (not the value specified on the VALUE declaration).

Chapter 20. VS Pascal Messages

The message prefixes identify when a message was generated—during compile time or run time, when debugging, or when running an EXEC or CLIST. To find a message explanation quickly, look up the message prefix in the first column of Figure 108. The page number indicates where the section for that message type starts; messages are arranged in numeric order within each section.

Note: Compiler messages without a prefix have an implied prefix of "AMPL".

Message Range	Message Type	See Page
(AMPL)0 to AMPL999S	Compiler Messages—Source Code Processing	216
AMPO001S to AMPO999S	Compiler Messages—Intermediate Code Optimization	240
AMPT001E to AMPT999S	Compiler Messages—Object Code Generation	241
AMPX011E to AMPX999S	Run-Time Messages	243
AMPD500 to AMPD544	Interactive Debugging Tool Messages	251
AMPE100E to AMPE206E	EXEC Messages	255
AMPC100W to AMPC102I	CLIST Messages	257

Figure 108. VS Pascal Message Summary

Compiler Messages—Source Code Processing

These messages are issued during the first compiler pass (syntactic and semantic analysis and intermediate code generation). They all have an implied prefix of AMPL. The action associated with each message class is shown below.

Note: All messages in this section are errors unless otherwise noted.

Message Class	Return Code	Object Code?	Other Actions
Informational	0	Yes	None
Warning	4	Yes	None
Error	8	No	None
Severe error	12	No	Compiler halts

0 UNSUPPORTED VS PASCAL FEATURE
Explanation: VS Pascal does not support the indicated construction.

Note: Do not report any problems caused by unsupported features to IBM.
Programmer Response: If you did not intend to use the feature, remove it from your code.
System Action: The compiler accepts this feature, but the results are unpredictable.

1 IDENTIFIER EXPECTED
Explanation: An identifier was expected but not found.
Programmer Response: Place a valid identifier at the indicated location.
System Action: The compiler assumes an identifier was found and continues compiling.

2 SOURCE CONTINUES AFTER END OF PROGRAM
Explanation: The compiler detected text after the logical end of the program. Mismatched BEGIN and END statements often cause this error.
Programmer Response: Either:

- Add END statements where appropriate.
- Delete the statement with the missing END statement.

System Action:

- If you were compiling a program unit, the compiler ignores the remaining code.
- If you were compiling a segment unit, the compiler compiles the remaining code.

3 "END" EXPECTED
Explanation: An END was expected but not found.
Programmer Response: Add an END at the intended location, or remove an unmatched BEGIN.
System Action: The compiler assumes an END statement was found and continues compiling.

4 CHARACTER IN QUOTED STRING IS NOT DISPLAYABLE
Explanation: The indicated character within a quoted string is not a valid, displayable EBCDIC character. If the string is printed on a device, the character might be interpreted as a control character, causing unpredictable results.

If you intend for the character to be a control character, you must make sure the string is in hexadecimal form.
Programmer Response: If you used the character unintentionally, correct the string.
System Action: The compiler accepts the character and continues compiling. This is a warning.

5 SYMBOL INVALID OR OUT OF CONTEXT
Explanation: The indicated symbol is not part of the syntax of the construct being scanned.
Programmer Response: Delete or change the symbol
System Action: The compiler attempts to recover and continues compiling.

6 EOF BEFORE LOGICAL END OF PROGRAM
Explanation: The compiler came to the end of the source program without finding the logical end of the program. This error is typically caused by:

- Mismatched BEGIN/END brackets
- Missing comment delimiters.

Programmer Response: Match BEGIN and END words properly. Make sure all comments are closed.
System Action: The compiler process all remaining input text and then stops.

7 "BEGIN" EXPECTED
Explanation: A BEGIN statement was expected but not found.
Programmer Response: Put a BEGIN where indicated.
System Action: The compiler assumes a BEGIN was found and continues compiling.

8 SEMICOLON ";" EXPECTED
Explanation: A semicolon was expected but not found.
Programmer Response: Put a semicolon where indicated.
System Action: The compiler assumes a semicolon was found and continues compiling.

9 ROUTINE MAY NOT BE PASSED TO FORTRAN SUBROUTINE

Explanation: The indicated subroutine is declared with the FORTRAN directive. The declaration of this routine also contains an argument that is a procedure or function parameter. You cannot pass procedures or functions to FORTRAN subroutines.
Programmer Response: If you did not intend to pass a routine as a parameter, pass the correct parameter. If you must pass a routine as a parameter, you must rewrite the FORTRAN subroutine in Pascal.
System Action: The compiler accepts the routine parameter and continues compiling.

10 NO CASE LABELS SPECIFIED

Explanation: A CASE statement appears with no case labels. A CASE statement cannot be empty or consist only of an OTHERWISE clause.
Programmer Response: Delete the CASE statement or add some alternatives to the CASE statement.
System Action: The compiler attempts to recover and continues compiling.

11 AMBIGUOUS PROCEDURE/FUNCTION SPECIFICATION

Explanation: The indicated routine is declared with an EXTERNAL, GENERIC, FORTRAN, MAIN, or REENTRANT routine directive. This routine was also declared as an ENTRY routine. This combination is contradictory.
Programmer Response: Delete the ENTRY pseudo-directive. VS Pascal does not support ENTRY.
System Action: The compiler attempts to recover and continues compiling.

12 MULTIPLY DECLARED LABEL

Explanation: The indicated label was previously declared within the current routine.
Programmer Response: Remove the duplicate declaration or use a different label.
System Action: The compiler attempts to recover and continues compiling.

13 LABEL IDENTIFIER EXPECTED

Explanation: Within a label declaration or GOTO statement, a label identifier is missing or invalid. A label identifier is either:

- An alphanumeric identifier
- An integer constant within the range 0 to 9999.

Programmer Response: Either:

- Use a correct label identifier.
- Remove the invalid declaration or GOTO.

System Action: The compiler attempts to recover and continues compiling.

14 THE CHARACTERS "$" AND "_" ARE NOT VALID IN STANDARD PASCAL

Explanation: This message can occur only when you specify the LANGLVL(ANSI83) compile-time option. An identifier name contains characters that are not recognizable in standard Pascal.
Programmer Response: Either:

- Compile with LANGLVL(EXTENDED).
- Delete the unrecognized characters.

System Action: The compiler accepts the characters and continues compiling. VS Pascal issues this message only when LANGLVL(ANSI83) is in effect. The message class is controlled by STDFLAG.

15 EQUAL SIGN "=" EXPECTED

Explanation: An equal sign was expected but not found.
Programmer Response: Put an equal sign in the indicated place.
System Action: The compiler assumes an equal sign was used and continues compiling.

16 IDENTIFIER REQUIRED TO BE A TYPE IN TAG FIELD SPECIFICATION

Explanation: The declaration of the tag field with a RECORD contains an error. The indicated identifier must represent the tag field's type, but the identifier is not declared as a type.
Programmer Response: Either

- Use a type identifier for the tag field.
- Use a different type of tag field specification.

System Action: The compiler attempts to recover and continues compiling.

17 COLON ":" EXPECTED

Explanation: A colon was expected but not found.
Programmer Response: Put a colon in the indicated place.
System Action: The compiler assumes a colon was used and continues compiling.

18 PARAMETERS ON FORWARDED ROUTINE NOT NECESSARY

Explanation: This error occurs when the body of a previously declared routine also declares formal parameters. This can occur when you declare routines with the FORWARD or EXTERNAL directives.

You include the formal parameters when you declare the header of the routine. When you declare the body of the routine, you must not specify any formal parameters.
Programmer Response: Delete the formal parameters.
System Action: The compiler attempts to compile the parameter list given.

19 FILES PASSED BY VALUE NOT PERMITTED

Explanation: The indicated formal value parameter is a file type or a type that contains a file. You can pass a file variable to a routine only by VAR or by CONST. You can never pass a file variable by value.
Programmer Response: Pass the parameter by VAR or CONST.
System Action: The compiler assumes the parameter was good and continues compiling.

20 FILES NOT PERMITTED IN STRUCTURED VALUE CONSTRUCTORS

Explanation: A type containing a file was used to construct a structured value. This is illegal, because files cannot be initialized.
Programmer Response: Either:

- Do not declare the type with a file in it.
- Use a different type as the structured value constructor.

System Action: The compiler assumes the type did not contain a file and continues compiling.

21 RIGHT PARENTHESIS ")" EXPECTED

Explanation: A right parenthesis, ")", was expected but not found.
Programmer Response: Add a right parenthesis in the indicated location.
System Action: The compiler assumes a right parenthesis was found and continues compiling.

22 FORWARDED ROUTINE CLASS CONFLICT

Explanation: Either:

- A procedure declaration was previously declared as a forwarded function.
- A function declaration was previously declared as a forwarded procedure.

Programmer Response: Either:

- Declare the routine consistently.
- Use a different name for one of the routines.

System Action: The compiler assumes a new declaration of the routine is occurring and continues compiling.

---

25 TYPE NOT NEEDED ON FORWARDED FUNCTION

Explanation: A function declaration that was previously forwarded must not specify a result type. VS Pascal assumes that the type specification appears on the declaration that contains the forwarding directive.

Programmer Response: Either:

- Remove the result type specification
- Do not declare the function with a directive initially.

System Action: The compiler compiles the result type.

---

26 MISSING TYPE SPECIFICATION FOR FUNCTION

Explanation: The indicated function header did not specify a return type.

Programmer Response: Either:

- Add a result type to the function declaration.
- Use a procedure instead of a function.

System Action: The compiler assumes a result type was given and continues compiling.

---

27 PROCEDURE/FUNCTION PREVIOUSLY FORWARDED

Explanation: The indicated routine declaration contains a forwarding directive (for example, FORWARD, EXTERNAL, MAIN, or REENTRANT). However, the routine was already previously forwarded.

Programmer Response: Remove one of the routine header declarations.

System Action: The compiler assumes the specified directive is correct and continues compiling.

---

28 ADDITIONAL ERRORS IN THIS LINE WERE NOT DIAGNOSED

Explanation: The indicated construct contains more errors, but the compiler did not diagnose these errors because of space considerations.

Programmer Response: Correct the errors on this line. The compiler might detect more errors on this line when you recompile the unit.

System Action: The compiler attempts to recover and continues compiling.

---

29 ILLEGAL HEXADECIMAL OR BINARY DIGIT

Explanation: The compiler detected either:

- An invalid hexadecimal digit within a hexadecimal constant specification with one of these forms:
  '...'X
  '...'XC
  '...'XG
  '...'XR.
- An invalid binary digit within a binary constant specification with the form:
  '...'B.

The following characters are valid hexadecimal digits:
0 through 9
A through F
a through f

The following characters are valid binary digits:
0
1.

Programmer Response: Use a valid digit or use a different base.

System Action: The compiler assumes a zero was intended and continues compiling.

---

30 UNIDENTIFIABLE CHARACTER

Explanation: The indicated character is not a valid token or is out of context.

Programmer Response: Remove the character from the source.

System Action: The compiler skips the character and continues compiling.

---

31 DIGIT EXPECTED

Explanation: A decimal digit was expected but missing at the indicated location.

Programmer Response: Place a digit where indicated.

System Action: The compiler attempts to recover and continues compiling. This omission is an error in most cases. However, when the LANGLVL(ANSI83) compile-time option is in effect, the compiler might issue this message with a different class, depending on the setting of STDFLAG, when the exponent character immediately follows the decimal point in a real constant.

---

34 END OF STRING NOT SEEN

Explanation: A string literal cannot cross a line boundary. This error is often caused by mismatched quotation marks.

Programmer Response: Place the string literal on a new line. If the literal is too large to fit on one line, break it up into multiple strings and concatenate the strings with the || operator. (Concatenation of string constants is performed at compile time.)

System Action: The compiler assumes the string was ended by a single quote at the end of the line and continues compiling.

---

37 STANDARD ROUTINES NOT PERMITTED AS PARAMETERS

Explanation: You cannot pass standard routines that generate in-line code as parameters to other routines. See "Routines That May Not Be Passed As Parameters" on page 277.

Programmer Response: Declare a routine that calls only the standard routine and pass this new routine as the parameter.

System Action: The compiler assumes the routine was legal to pass as a parameter and continues compiling.

---

38 VARIABLE MUST BE OF TYPE FILE

Explanation: The indicated variable must be a FILE type.

Programmer Response: Use a file variable where indicated.

System Action: The compiler assumes a file was specified and continues compiling.

---

39 MUST BE OF TYPE TEXT

Explanation: The indicated variable must be a TEXT type.

Programmer Response: Use a TEXT file where indicated.

System Action: The compiler assumes a TEXT file was specified and continues compiling.

40 REQUIRED PARAMETERS ARE MISSING

Explanation: The indicated read or write procedure is missing a required parameter. Follow these procedures:

- READ requires at least one variable to read data into.
- WRITE requires at least one expression to write data from.
- READSTR requires a string expression and at least one variable.
- WRITESTR requires a string variable and at least one expression.

Programmer Response: Specify the correct parameters for the routine.

System Action: The compiler attempts to recover and continues compiling.

41 COMMA "," EXPECTED

Explanation: A comma was expected but not found.

Programmer Response: Use a comma where indicated.

System Action: The compiler attempts to recover and continues compiling.

42 USER DEFINED SCALARS NOT PERMITTED

Explanation: Expressions that are user-defined enumerated types cannot be directly read from or written to a text file.

Programmer Response: Either:

- Use a CASE statement to assign the name of the value of the enumerated expression into a temporary string variable and write that string out.
- Write the ordinal value of the enumerated expression out.

System Action: The compiler attempts to recover and continues compiling.

43 OPERAND OF READ/WRITE NOT OF A VALID TYPE

Explanation: Any parameter passed to the procedures READ or WRITE (TEXT file) must be compatible with one of the following types:

- INTEGER
- REAL
- SHORTREAL
- CHAR
- GCHAR
- BOOLEAN (WRITE only)
- STRING
- GSTRING
- PACKED ARRAY[1..*n*] OF CHAR (where *n* is a positive integer constant)
- PACKED ARRAY[1..*n*] OF GCHAR (where *n* is a positive integer constant).

Programmer Response: Ensure the parameter passed is the intended one. If it is, either:

- Convert the parameter to an appropriate type.
- Do not pass the parameter.

System Action: The compiler attempts to recover and continues compiling.

44 FIELD LENGTH MUST BE INTEGER

Explanation: 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.

Programmer Response: Use an integer expression.

System Action: The compiler attempts to recover and continues compiling.

45 SET CONTAINS CONSTANT MEMBER(S) WHICH ARE OUT OF RANGE

Explanation: The indicated set constant contains members that 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 *)
                      (* produces error 45. *)
END;
```

This error can also occur when a set constant is being passed as a parameter.

Programmer Response: Either:

- Extend the set bounds.
- Use proper values in the set constant.

System Action: The compiler attempts to recover and continues compiling.

46 SECOND FIELD LENGTH APPLICABLE ONLY TO REAL DATA

Explanation: In the procedure WRITE (TEXT file case), only expressions of type REAL (or SHORTREAL) are permitted to have two length field qualifications.

Programmer Response: Either:

- Use a real expression.
- Delete the second length field.

System Action: The compiler attempts to recover and continues compiling.

47 ARRAY REFERENCE CONTAINS TOO MANY SUBSCRIPTS

Explanation: An array variable of dimension 'n' is being subscripted with more than 'n' subscripts.

Programmer Response: Ensure the array being subscripted is the correct one. If it is, use the correct number of subscripts.

System Action: The compiler attempts to recover and continues compiling.

48 ASSOCIATED VARIABLE OF SUBSCRIPT MUST BE OF AN ARRAY TYPE

Explanation: An attempt is being made to subscript a variable that is not declared as an array.

Programmer Response: Ensure the variable being subscripted is the one intended. If it is, either:

- Remove the subscript.
- Declare the variable as an array.

System Action: The compiler attempts to recover and continues compiling.

49 EXPRESSION MUST BE OF A SIMPLE SCALAR TYPE

Explanation: Because of the context in which it is being used, the indicated expression must be a simple scalar type.

Programmer Response: Use a simple scalar expression.

System Action: The compiler attempts to recover and continues compiling.

50 NO MAXIMUM LENGTH SPECIFIED ON STRING TYPE; 255 ASSUMED

Explanation: A type definition of the form "STRING" does not contain a length specification to indicate the maximum length of the string variable. The default length is 255.
Programmer Response: Explicitly specify the maximum length desired.
System Action: The compiler attempts to recover and continues compiling. This is a warning.

51 VARIABLE MUST BE OF A POINTER TYPE

Explanation: The indicated variable is used as a pointer. However, the variable is not declared as a pointer.
Programmer Response: Ensure the variable being used is the correct one. If it is, either:

- Redeclare the variable as a pointer.
- Do not use the variable as if it were a pointer.

System Action: The compiler attempts to recover and continues compiling.

52 CORRESPONDING VARIANT DECLARATION MISSING

Explanation: Within a call to the procedure NEW or to the function SIZEOF, either:

- The indicated tag field specification fails to correspond to a variant within the associated record variable.
- The associated variable was not of a record type.

Programmer Response: If the variant being used was correct, include it in the record declaration.
System Action: The compiler attempts to recover and continues compiling.

54 EXPRESSION MUST BE NUMERIC

Explanation: Expressions that are prefixed with a sign ('+' or '-') must be of a type that is compatible with INTEGER or REAL. This also applies to expressions that are operands of such predefined functions as ABS and SQR.
Programmer Response: Either:

- Use a numeric expression.
- Delete the sign.

System Action: The compiler attempts to recover and continues compiling.

55 EXPRESSION MUST BE OF TYPE REAL

Explanation: 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 (or SHORTREAL) as a parameter.
Programmer Response: Ensure the parameter is a real value.
System Action: The compiler attempts to recover and continues compiling.

56 EXPRESSION MUST BE OF TYPE INTEGER

Explanation: The indicated expression must be of a type that is compatible with INTEGER.
Programmer Response: Ensure the expression is an integer.
System Action: The compiler attempts to recover and continues compiling.

57 PARAMETER TYPE DOES NOT MATCH FORMAL PARAMETER

Explanation: Within a procedure or function call, an expression or variable being passed as an actual parameter is of a type that is not compatible with the corresponding formal parameter.
Programmer Response: Ensure the actual parameter was the one intended. If it is, either:

- Convert the the actual parameter to the correct type.
- Modify the formal parameter.

System Action: The compiler attempts to recover and continues compiling.

58 EXPRESSION MUST BE A VARIABLE

Explanation: An erroneous attempt was made to pass a non-variable as an actual parameter to a routine that expects a pass-by-VAR parameter.
Programmer Response: Either:

- Use a variable.
- Pass the parameter by value or CONST.

System Action: The compiler attempts to recover and continues compiling.

59 NUMBER OF PARAMETERS DOES NOT AGREE

Explanation: Within a procedure or function call, the number of parameters being passed does not correspond with the number required.
Programmer Response: Ensure the routine being called was the one intended. If it is, either:

- Pass the correct number of actual parameters.
- Modify the routine declaration by changing the number of formal parameters.

System Action: The compiler attempts to recover and continues compiling.

60 LEFT PARENTHESIS "(" EXPECTED

Explanation: A left parenthesis, "(", was expected but not found.
Programmer Response: Use a left parenthesis where indicated.
System Action: The compiler attempts to recover and continues compiling.

61 CONSTANT EXPECTED

Explanation: At the place indicated, a constant was expected but is missing.
Programmer Response: Use a constant where indicated.
System Action: The compiler attempts to recover and continues compiling.

62 TYPE SPECIFICATION EXPECTED

Explanation: At the place indicated, a type definition is expected but is missing.
Programmer Response: Use a type specification where indicated.
System Action: The compiler attempts to recover and continues compiling.

63 RANGE SPECIFICATION SYMBOL ".." EXPECTED

Explanation: A range specification symbol was expected but not found.
Programmer Response: Use a range symbol where indicated.
System Action: The compiler attempts to recover and continues compiling.

64 EXPRESSION'S TYPE IS INCORRECT OR INCOMPATIBLE WITHIN CONTEXT

Explanation: This error is caused by a number of reasons, including:

- A unary or binary operator is being applied to an expression that is of a type that is not valid for the operator.
- A binary operator is being used on two expressions of incompatible types.
- The parameters of the MIN or MAX functions are of incompatible types.
- Members of a set constructor have inconsistent types.

Programmer Response: Ensure all types are compatible where indicated.
System Action: The compiler attempts to recover and continues compiling.

65 SUBRANGE LOWER BOUND IS GREATER THAN UPPER BOUND

Explanation: The lower bound of a subrange was greater than the specified upper bound.
Programmer Response: Adjust one of the bounds so that the lower bound is less than or equal to the upper bound.
System Action: The compiler attempts to recover and continues compiling.

66 ASSIGNMENT TO POINTER QUALIFIED VARIANT RECORD UNSAFE

Explanation: The indicated statement attempts an assignment to an entire pointer-qualified record variable with variant fields. Such an assignment is dangerous under VS Pascal. This message is issued because the pointer-qualified record might have been allocated with a size that is specific to its active variant.

Here is an example of the error:

```
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
```

Programmer Response: Ensure the value being assigned is at least as long as the variable being assigned into.
System Action: The compiler generates code to perform the assignment and continues compiling. This is a warning.

67 REAL TYPE NOT VALID HERE

Explanation: The indicated expression is of type REAL or SHORTREAL. An expression of this type is not valid within the associated context.
Programmer Response: Do not use a REAL or SHORTREAL value here.
System Action: The compiler attempts to recover and continues compiling.

68 "OF" EXPECTED

Explanation: An OF was expected but not found.
Programmer Response: Use the reserved word OF where indicated.
System Action: The compiler attempts to recover and continues compiling.

69 TAG CONSTANT DOES NOT MATCH TAG FIELD TYPE

Explanation: Within a RECORD definition, a variant tag is being defined that 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 that is of a type that is not compatible with the corresponding tag field type of an associated record variable.
Programmer Response: Either:

- Use a valid tag constant.
- Change the variant definition to use a different type.

System Action: The compiler attempts to recover and continues compiling.

70 DUPLICATE VARIANT FIELD

Explanation: Within a RECORD definition, a variant tag is being defined more than once.
Programmer Response: Delete or change one of the duplicate variant tags.
System Action: The compiler attempts to recover and continues compiling.

71 NOT APPLICABLE TO "PACKED" QUALIFIER

Explanation: The indicated type definition was qualified with the word "PACKED". Such a qualification within the associated context is not valid.
Programmer Response: Remove the PACKED reserved word from the type declaration.
System Action: The compiler attempts to recover and continues compiling.

72 LEFT BRACKET "[" EXPECTED

Explanation: A left bracket, "[", was expected but not found. A "(." is also acceptable.
Programmer Response: Use a left bracket where indicated.
System Action: The compiler attempts to recover and continues compiling.

74 RIGHT BRACKET "]" EXPECTED

Explanation: A right bracket, "]", was expected but not found. A ".)" is also acceptable.
Programmer Response: Use a right bracket where indicated.
System Action: The compiler attempts to recover and continues compiling.

75 LENGTH QUALIFIER APPLICABLE ONLY TO STRING TYPE

Explanation: A length qualifier was applied to a non-STRING type. STRINGs are the only types that can have length qualifiers.
Programmer Response: Either:

- Use a STRING.
- Remove the length qualifier.

System Action: The compiler attempts to recover and continues compiling.

76 FILE OF FILES NOT SUPPORTED

Explanation: A file cannot contain a file or a type that contains a file.

Programmer Response: Ensure the type of the file is correct. If it is, either:

- Do not declare a file of this type.
- Redeclare the type so that it does not contain a file.

System Action: The compiler attempts to recover and continues compiling.

77 ILLEGAL REFERENCE OF FUNCTION NAME

Explanation: The indicated identifier is the name of a function. This identifier is being used incorrectly. This error can occur when you try to assign a result to:

- A function outside of the function
- A function that is either predefined or a formal parameter.

Programmer Response:

System Action: The compiler attempts to recover and continues compiling.

78 SUBSCRIPT TYPE NOT COMPATIBLE WITH INDEX TYPE

Explanation: The indicated subscript expression is not of a type that is compatible with the declared subscript type for the array.

Programmer Response: Either:

- Ensure the expression returns a type compatible with the index type.
- Change the index type.

System Action: The compiler attempts to recover and continues compiling.

79 ASSOCIATED VARIABLE MUST BE OF A RECORD TYPE

Explanation: 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.

Programmer Response: Ensure a record is used where indicated.

System Action: The compiler attempts to recover and continues compiling.

80 RECORD FIELD QUALIFIER NOT DEFINED

Explanation: The indicated identifier was not defined as a record field.

Programmer Response: Do one of the following:

- Use a different record name.
- Use a different qualifier name.
- Do not use the period qualification.

System Action: The compiler assumes the variable was a record and continues compiling.

81 QUALIFIED VARIABLE NOT A RECORD

Explanation: A variable was qualified that was not a record.

Programmer Response: Either:

- Use a record variable name to the left of the period.
- Do not qualify the variable.

System Action: The compiler assumes the variable was a record and continues compiling.

82 ASSOCIATED VARIABLE MUST BE OF A POINTER OR FILE TYPE

Explanation: The indicated arrow-qualified variable is not of a pointer or file type.

Programmer Response: Either:

- Use a valid pointer or file name.
- Delete the pointer dereference symbol.

System Action: The compiler attempts to recover and continues compiling.

83 SET ELEMENT OUT OF RANGE

Explanation: The indicated set member of a set constructor exceeds the allowed range for the set.

Programmer Response: Either:

- Use a valid set member.
- Expand the range of the set.

System Action: The compiler attempts to recover and continues compiling.

84 EXPRESSION MUST BE OF A SET TYPE

Explanation: The indicated expression is required to be of a set type in the context in which it is being used.

Programmer Response: Ensure the expression is a constant expression resulting in an integer value.

System Action: The compiler attempts to recover and continues compiling.

85 MUST BE POSITIVE INTEGER CONSTANT

Explanation: The indicated expression fails to evaluate to a positive integer constant. Because of the context in which it is used, the expression must evaluate to a positive integer expression.

Programmer Response: Ensure the expression evaluates to a positive integer.

System Action: The compiler attempts to recover and continues compiling.

86 LEAVE/CONTINUE NOT WITHIN LOOP

Explanation: The indicated LEAVE or CONTINUE statement does not reside within a loop construct.

Programmer Response: Either:

- Delete the LEAVE or CONTINUE.
- Make sure the LEAVE or CONTINUE is in a loop.

System Action: The compiler attempts to recover and continues compiling.

87 ASSIGNMENT SYMBOL ":=" EXPECTED

Explanation: An assignment symbol, ":=", was expected but not found.

Programmer Response: Place an assignment symbol where indicated.

System Action: The compiler attempts to recover and continues compiling.

89 TEXT FILES MAY NOT BE UPDATED

Explanation: An attempt was made to open a text file for updating. Only record files can be updated.

Programmer Response: Use an identifier with a record file type.

System Action: The compiler attempts to recover and continues compiling.

90 LABEL NOT DECLARED

Explanation: The indicated label did not appear in a LABEL declaration.

Programmer Response: Either:

- Declare the label.
- Use the correct label identifier.

System Action: The compiler attempts to recover and continues compiling.

91 MAXIMUM LENGTH OF STRING VARIABLE DOES NOT MATCH FORMAL PARAMETER

Explanation: 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.

Here is an example this error:

```
PROCEDURE XYZ(VAR S: STRING(100));
          EXTERNAL;
  VAR T: STRING(50);
    BEGIN
      .
      .
      XYZ(T); (* ERROR: The declared length *)
      .       (* of T does not match that   *)
      .       (* of parameter S.            *)
    END
```

Programmer Response: Either:

- Use conformant string parameters.
- Change the declaration of one of the parameters.

System Action: The compiler attempts to recover and continues compiling.

92 "THEN" EXPECTED

Explanation: A THEN was expected but not found.

Programmer Response: Put a THEN where indicated.

System Action: The compiler attempts to recover and continues compiling.

93 REDUNDANT CASE ALTERNATIVE

Explanation: The indicated CASE statement label is equal to a previous label within the same CASE statement.

Programmer Response: Either:

- Delete one of the duplicate CASE labels.
- Change one of them to the correct value.

System Action: The compiler attempts to recover and continues compiling.

94 REQUIRED LENGTH EXPRESSION MISSING FOR DYNAMIC STRING ALLOCATION

Explanation: A pointer variable declared with the type STRINGPTR is being allocated with the NEW procedure, but the required length expression is missing.

Programmer Response: Add an expression to the call to NEW that specifies the length of the string desired.

System Action: The compiler attempts to recover and continues compiling.

95 "UNTIL" EXPECTED

Explanation: An UNTIL was expected but not found.

Programmer Response: Put an UNTIL where indicated.

System Action: The compiler attempts to recover and continues compiling.

96 "DO" EXPECTED

Explanation: A DO was expected but not found.

Programmer Response: Put a DO where indicated.

System Action: The compiler attempts to recover and continues compiling.

97 FOR-LOOP INDEX MUST BE SIMPLE LOCAL VARIABLE

Explanation: A FOR loop variable must be declared as a simple automatic (VAR) variable, local to the routine in which the FOR loop resides. This means that the loop control variable cannot be:

- Cannot be a DEF, REF, or STATIC variable
- Cannot be a parameter to the routine
- Can be part of a structure.

The indicated FOR loop variable did not meet this criteria.

Programmer Response: Make the loop control variable a local VAR variable.

System Action: The compiler attempts to recover and continues compiling.

98 "TO" EXPECTED

Explanation: A TO was expected but not found.

Programmer Response: Put a TO where indicated.

System Action: The compiler attempts to recover and continues compiling.

99 LABEL PREVIOUSLY DEFINED

Explanation: The indicated label identifier was previously defined within the associated routine.

Programmer Response: Change or delete one of the duplicate label identifiers.

System Action: The compiler attempts to recover and continues compiling.

103 EXPRESSION MUST BE OF TYPE BOOLEAN

Explanation: The indicated expression that is associated with an IF, ASSERT, WHILE, or REPEAT statement must represent a condition. Conditional expressions are of type BOOLEAN. The indicated expression failed to meet this criteria.

Programmer Response: Ensure the expression results in a BOOLEAN value.

System Action: The compiler attempts to recover and continues compiling.

104 CONSTANT OUT OF RANGE

Explanation: The indicated constant expression evaluated to a value that is outside the range required by its context.

Programmer Response: Either:

- Use a valid constant.
- If possible, expand the range of constants allowed.

System Action: The compiler attempts to recover and continues compiling.

105 IDENTIFIER WAS PREVIOUSLY DECLARED

Explanation: The indicated identifier within a declaration was previously declared within the same lexical scope.

Programmer Response: Either:

- Change one of the declarations to use a different name.
- Delete one of the declarations.

System Action: The compiler attempts to recover and continues compiling.

106 UNDECLARED IDENTIFIER

Explanation: The indicated identifier being referenced was not declared. This can occur when using an unqualified record field name outside of a WITH statement.

Programmer Response: Do one of the following:

- Use the correct identifier.
- Declare the indicated identifier.
- Qualify the identifier with a record value if the identifier is a field name.

System Action: The compiler attempts to recover and continues compiling.

107 IDENTIFIER IS NOT IN PROPER CONTEXT

Explanation: The indicated identifier is being used in a way that is not consistent with how it was declared.

Programmer Response: Use an identifier that is valid in this context.

System Action: The compiler attempts to recover and continues compiling.

109 CASE LABEL TAG OF WRONG TYPE

Explanation: The value of the indicated CASE statement label is not of a type that can conform to the CASE statement indexing expression.

Programmer Response: Either:

- Use a valid case label.
- Modify the type of the CASE expression.

System Action: The compiler attempts to recover and continues compiling.

110 LOOP WILL NEVER EXECUTE

Explanation: The indicated FOR loop will not execute at run time. The compiler has determined that the terminating condition for the loop is unconditionally true.

Programmer Response: Correct the error if this condition was unintentional.

System Action: The compiler attempts to recover and continues compiling. This is a warning.

111 LOOP RANGE EXCEEDS RANGE OF INDEX

Explanation: 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.

Programmer Response: Either:

- Enlarge the range of the variable.
- Change the loop bounds.

System Action: The compiler attempts to recover and continues compiling. This is a warning.

112 "PROGRAM" HEADER MISSING

Explanation: The PROGRAM symbol was expected but not found.

Programmer Response: Ensure the first keyword in the program is PROGRAM.

System Action: The compiler attempts to recover and continues compiling.

113 NESTED COMMENT DELIMITER FOUND

Explanation: A starting comment symbol was detected within a comment opened by the same starting symbol. This error often occurs because of mismatched comment delimiters.

Programmer Response: Either:

- Delete the nested comment delimiter.
- Terminate the original comment.

System Action: The compiler attempts to recover and continues compiling. This is a warning.

119 FIRST PARAMETER OF READSTR/WRITESTR MUST BE OF TYPE STRING

Explanation: The first parameter of the indicated READSTR or WRITESTR procedure was not a STRING.

Programmer Response: Ensure the first parameter is a string.

System Action: The compiler attempts to recover and continues compiling.

120 STRING CONSTANT REQUIRES TRUNCATION

Explanation: The indicated string constant is being assigned to a variable or being passed to a routine. This string constant requires truncation because of its excessive length. Implicit truncation of strings is not permitted.

Programmer Response: Either:

- Use a larger string in the assignment or as the formal parameter.
- Explicitly truncate the string being assigned or passed.

System Action: The compiler attempts to recover and continues compiling.

121 DECLARATION OUT OF ORDER: LABEL, CONST, TYPE, VAR, ROUTINE

Explanation: This message can be produced when the LANGLVL(ANSI83) compile-time 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:

1. Labels (LABEL)
2. Constants (CONST)
3. Types (TYPE)
4. Variables (VAR)
5. Routines (PROCEDURE/FUNCTION)

Programmer Response: Fix the order of the declarations if this was unintentional, or consider compiling with LANGLVL(EXTENDED).

System Action: The compiler attempts to recover and continues compiling. VS Pascal issues this message only when LANGLVL(ANSI83) is in effect. The message class is controlled by STDFLAG.

122 "OTHERWISE" CLAUSE WITHOUT ASSOCIATED CASE STATEMENT

Explanation: The indicated OTHERWISE statement is not within the context of a CASE statement.

Programmer Response: Ensure the OTHERWISE statement is within a CASE statement or delete the OTHERWISE statement.

System Action: The compiler attempts to recover and continues compiling.

124 CONSTRUCT OR OPERATION IS NOT IN STANDARD PASCAL

Explanation: This message may be produced when the LANGLVL(ANSI83) compile-time option is specified. The indicated language construct or arithmetic operation is not supported in standard Pascal, but is a VS Pascal language extension.
Programmer Response: Either compile with LANGLVL(EXTENDED) or remove the non-standard feature.
System Action: The compiler attempts to recover and continues compiling. VS Pascal issues this message only when LANGLVL(ANSI83) is in effect. The message class is controlled by STDFLAG.

125 REAL TO INTEGER CONVERSION NOT VALID

Explanation: 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.
Programmer Response: Convert the REAL or SHORTREAL value to an integer value using TRUNC or ROUND or use a REAL or SHORTREAL instead of an integer.
System Action: The compiler accepts the extension and continues compiling.

126 TYPES NOT CONFORMABLE IN ASSIGNMENT

Explanation: The indicated assignment statement attempts to assign an expression of a particular type to a variable of an incompatible type.
Programmer Response: Ensure the types of the variable and the expression are compatible.
System Action: The compiler attempts to recover and continues compiling.

127 FILE VARIABLE ASSIGNMENT NOT PERMITTED

Explanation: The left side of the indicated assignment statement is a variable of a file type or of a type that contains a file. Assignment to file variables is not permitted.
Programmer Response: Ensure the type of file is correct. If it is, either do not assign a variable of this type, or redeclare the type so that it does not contain a file.
System Action: The compiler attempts to recover and continues compiling.

128 NOT COMPILE-TIME COMPUTABLE

Explanation: The indicated expression fails to be a constant expression that can be evaluated at compile time.
Programmer Response: Use a variable to hold the value of the expression, or make the expression computable at compile time.
System Action: The compiler attempts to recover and continues compiling.

129 ASSIGNMENT TO "CONST" PARAMETER INVALID

Explanation: The indicated variable declared as a formal CONST parameter within a particular routine cannot be modified by an assignment.
Programmer Response: Ensure that the parameter being modified is the correct one, and use a pass-by-VAR or pass-by-value parameter if it is correct.
System Action: The compiler attempts to recover and continues compiling.

130 ASSIGNMENT TO FOR-LOOP INDEX INVALID

Explanation: The indicated variable that is being used as a FOR loop index cannot be modified by an assignment within the FOR loop statement.
Programmer Response: Ensure the variable being modified is the correct one, and use another variable to hold the index if it is correct.
System Action: The compiler attempts to recover and continues compiling.

131 PASSING "CONST" PARAMETER BY VAR INVALID

Explanation: The indicated variable declared as a formal CONST parameter cannot be modified by being passed as an actual VAR parameter to a routine.
Programmer Response: Ensure that the parameter being passed by VAR is the correct one, and, if it is correct, either pass the parameter by value or by CONST to the called routine or change the calling routine's parameter to a pass-by-value or pass-by-VAR parameter.
System Action: The compiler attempts to recover and continues compiling.

132 PASSING FOR-LOOP INDEX BY VAR INVALID

Explanation: The indicated variable that is being used as a FOR loop index cannot be modified by being passed as an actual VAR parameter to a routine.
Programmer Response: Pass the index by value or CONST instead of by VAR.
System Action: The compiler attempts to recover and continues compiling.

133 REFER-BACK TAGFIELD MUST NOT BE TYPED

Explanation: The indicated tag field specification within a record definition was found to reference a previous field within the record. Such refer-back references cannot contain a type reference.
Programmer Response: Either remove the type from the tag field specification or use a different name for the tag field.
System Action: The compiler attempts to recover and continues compiling.

137 PASSING PACKED RECORD FIELD BY VAR NOT VALID

Explanation: This message may be produced when the LANGLVL(ANSI83) 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.
Programmer Response: If this is intentional, you might want to compile with the LANGLVL(EXTENDED) option. Otherwise, remove the PACKED specification from the record whose field is being passed, assign the field to an unpacked variable and pass it, or pass the field by value.
System Action: The compiler passes the field as a packed record field and continues compiling. VS Pascal issues this message only when the LANGLVL(ANSI83) compile-time option is in effect. The message class is controlled by STDFLAG.

138 PASSING SPACE COMPONENT BY VAR NOT VALID

Explanation: This message may be produced when the LANGLVL(ANSI83) compile-time option is in effect. Standard Pascal requires that actual VAR parameters be properly aligned, but this is not necessarily the case with a SPACE component. The indicated parameter is a component of a SPACE variable that is being passed as a VAR parameter.
Programmer Response: Either do not use a SPACE variable in LANGLVL(ANSI83) or compile with LANGLVL(EXTENDED).
System Action: The compiler attempts to recover and continues compiling. This is a warning.

139 PASSING PACKED ARRAY ELEMENT BY VAR NOT VALID

Explanation: 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.
Programmer Response: If this is intentional, you may want to compile with the LANGLVL(EXTENDED) option. Otherwise, remove the PACKED specification from the array whose element is being passed, copy the packed array into an unpacked variable and pass it, or pass the element by value.
System Action: The compiler passes the array element as a packed array element if the element was byte aligned and continues compiling. This is an error if the element was not byte aligned. Otherwise, this is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

140 ACTUAL AND FORMAL PARAMETER SIZES DIFFER

Explanation: The actual parameter being passed by VAR did not have the same storage size as the formal parameter. This could lead to storage overlays, and thus is illegal. This can happen when the packing differs between the actual and formal parameters, or when REAL and SHORTREAL types are mixed.
Programmer Response: Ensure the actual and formal parameters require the same amount of storage or pass the parameter by value or CONST.
System Action: The compiler assumes the types had the same sizes and continues compiling.

141 SYMBOL NOT RECOGNIZABLE IN STANDARD PASCAL

Explanation: This message may result when the LANGLVL(ANSI83) compiler option is specified. The indicated symbol (or operator) is not supported in Standard Pascal. The symbol is part of a construct that is a VS Pascal language extension.
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED); otherwise, do not use the indicated symbol.
System Action: The compiler accepts the symbol and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

142 VARIABLE MUST BE AN ARRAY VARIABLE

Explanation: The indicated variable is required to be of an array type, but such is not the case.
Programmer Response: Use an array variable.
System Action: The compiler attempts to recover and continues compiling.

143 OFFSET QUALIFIED FIELD NOT ON PROPER BOUNDARY

Explanation: The indicated field in a record definition is qualified with an offset that is not consistent with the boundary requirement of the field's type.
Programmer Response: Either ensure the offset specified is valid for the field type or declare the record PACKED.
System Action: The compiler attempts to recover and continues compiling.

144 OFFSET QUALIFICATION VALUE IS TOO SMALL

Explanation: The indicated field in a record definition is qualified with an offset that either causes an overlap with a previous field within the record or is an illegal (negative) offset.
Programmer Response: Make the offset a positive number that does not overlap any previous fields.
System Action: The compiler attempts to recover and continues compiling.

145 TYPE MUST BE CHAR OR PACKED ARRAY OF CHAR

Explanation: The indicated expression is required by its context to be of type CHAR or PACKED ARRAY[1..n] OF CHAR.
Programmer Response: Use the correct type.
System Action: The compiler attempts to recover and continues compiling.

146 VARIABLES OF TYPE POINTER ARE NOT PERMITTED

Explanation: The special type "POINTER" can be applied only to a formal parameter of a routine.
Programmer Response: Do not use the "POINTER" type.
System Action: The compiler attempts to recover and continues compiling.

147 IDENTIFIER WAS NOT DECLARED AS FUNCTION

Explanation: The indicated identifier is used as though it is a function name, but is not declared as such. This can be caused by having a left parenthesis following an identifier that is not a function or by using a type name that is not valid in a structured constant or scalar conversion.
Programmer Response: Use a valid function or type identifier.
System Action: The compiler attempts to recover and continues compiling.

148 MISSING PERIOD "." ASSUMED

Explanation: The indicated expression performs a comparison operation on two entities for which such comparison is not allowed. Except for strings, variables of structured types cannot be directly compared with each other. The only valid comparison operators for sets are '=', '<>', '<=', and '>='.
Programmer Response: Use a valid comparison operator, or write a routine to do the comparison.
System Action: The compiler attempts to recover and continues compiling.

149 NOT A VALID COMPARISON OPERATION

Explanation: The indicated expression performs a comparison operation on two entities for which such comparison is not allowed. Except for strings, variables of structured types cannot be directly compared with each other. The only valid comparison operators for sets are '=', '<>', '<=', and '>='.

Programmer Response: Use a valid comparison operator, or write a routine to do the comparison.

System Action: The compiler attempts to recover and continues compiling.

---

150 ENTRY ROUTINES MUST BE AT THE OUTERMOST NESTING LEVEL

Explanation: A routine that is to be called from another module is nested within another routine. This type of nesting is not allowed. Such routines must be declared at the outermost nesting level.

Programmer Response: Move the routine to the outermost level, or do not make the routine an entry point.

System Action: The compiler attempts to recover and continues compiling.

---

152 CHECKING ERROR WILL INEVITABLY OCCUR AT EXECUTION TIME

Explanation: This error indicates that the compiler has detected a condition related to a particular construct that will cause a run-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. *)
```

Programmer Response: Ensure any variables have appropriate bounds or change the statement to avoid the error.

System Action: The compiler attempts to recover and continues compiling.

---

153 LBOUND/HBOUND DIMENSION NUMBER IS INVALID FOR VARIABLE

Explanation: An invalid dimension number was used as the second parameter for LBOUND or HBOUND.

Programmer Response: Fix the dimension number or the array declaration.

System Action: The compiler attempts to recover and continues compiling.

---

154 LOW BOUND OF SUBSCRIPT RANGE IS TOO LARGE IN MAGNITUDE

Explanation: The indicated array definition has an illegal subscript range that causes addressing code to be outside the range of the target machine's capability.

Programmer Response: This error should not occur. If it does, consult the *VS Pascal Diagnosis Guide and Reference* for help.

System Action: The compiler attempts to recover and continues compiling.

---

156 LENGTH FIELDS NOT APPLICABLE TO NON-TEXT FILES

Explanation: A record file READ or WRITE contains a length qualified parameter. Length specifications have no meaning in record file I/O.

Programmer Response: Either remove the length specification or use a text file.

System Action: The compiler attempts to recover and continues compiling.

---

157 STRING VARIABLE IS SMALLER THAN FILE COMPONENT

Explanation: 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.

Programmer Response: Fix the string or file declaration.

System Action: The compiler attempts to recover and continues compiling.

---

158 COMMENT NOT CLOSED

Explanation: A comment was opened but never closed, and the rest of the source program was read in as a comment.

Programmer Response: Add an appropriate closing comment delimiter, or remove the opening comment delimiter and any non-program text.

System Action: The compiler stops executing, possibly after issuing other diagnostics.

---

159 RECURSIVE TYPE REFERENCE IS NOT PERMITTED

Explanation: The compiler detected a degenerate TYPE declaration of one of the following forms:

Form I:

```
TYPE X = X;
```

Form II:

```
TYPE X = RECORD
         .
         .
         F: X;
         .
         .
END
```

Programmer Response: Fix the type declaration.

System Action: The compiler attempts to recover and continues compiling.

---

160 THIS SET OPERATION WILL ALWAYS PRODUCE THE NULL SET

Explanation: Two disjoint sets are being intersected. The result will always be the null set []. Here is an example of the error:

```
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
```

Programmer Response: Ensure the null set is what is desired in this location.

System Action: The compiler accepts the expression and continues compiling. This is a warning.

---

161 ELSE CLAUSE WITHOUT ASSOCIATED IF STATEMENT

Explanation: An 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 (;).
Programmer Response: Remove the semicolon from the THEN clause, or ensure the conditional statement nesting is correct.
System Action: The compiler assumes the ELSE statement is part of an IF statement and continues compiling.

---

162 MUST BE AN UNPACKED ARRAY

Explanation: The indicated array variable is erroneously declared as PACKED when the context requires it to be unpacked.
Programmer Response: Remove the PACKED keyword from the array declaration or use a different array variable.
System Action: The compiler attempts to recover and continues compiling.

---

163 MUST BE A PACKED ARRAY

Explanation: The indicated array variable should have been declared as PACKED, but was not.
Programmer Response: Add the PACKED keyword to the array declaration or use a different array variable.
System Action: The compiler attempts to recover and continues compiling.

---

164 UNRECOGNIZABLE PROCEDURE/FUNCTION DIRECTIVE

Explanation: The indicated identifier was interpreted as a procedure or function directive but was not recognizable. The following are the only recognizable directives:

FORWARD
EXTERNAL
GENERIC
FORTRAN
MAIN
REENTRANT

Programmer Response: Use one of the above directives, or, if no directive was intended, ensure the routine declaration is syntactically correct.
System Action: The compiler attempts to recover and continues compiling.

---

165 FORTRAN SUBROUTINES MAY NOT BE PASSED AS PARAMETERS

Explanation: Only VS Pascal routines may be passed as parameters; FORTRAN subroutines may not.
Programmer Response: Define a VS Pascal procedure that does nothing more than call the FORTRAN subroutine, and then pass the VS Pascal procedure in place o the FORTRAN subroutine. Otherwise, replace the FORTRAN directive with the EXTERNAL directive if possible.
System Action: The compiler attempts to recover and continues compiling.

---

166 FORTRAN SUBROUTINE PARAMETERS MAY NOT BE PASSED BY VALUE

Explanation: All formal parameters of a FORTRAN subroutine must be passed by reference: either by VAR or by CONST.
Programmer Response: Declare the indicated parameter as pass-by-VAR or pass-by-CONST.
System Action: The compiler attempts to recover and continues compiling.

---

167 FORTRAN FUNCTIONS MAY RETURN ONLY SCALAR VALUES

Explanation: A FORTRAN function may only return values that are scalars (including floating point).
Programmer Response: Return a scalar result or call an EXTERNAL function which can return non-scalar results.
System Action: The compiler attempts to recover and continues compiling.

---

168 GENERIC ROUTINE PARAMETERS MAY NOT BE PASSED BY VALUE

Explanation: All actual parameters of a GENERIC routine must be passed by reference: either by VAR or by CONST.
Programmer Response: Add the VAR or CONST reserved word before the actual parameter.
System Action: The compiler accepts the value parameter and continues compiling.

---

169 ROUTINES MAY NOT BE PASSED TO GENERIC ROUTINES

Explanation: The indicated reserved word attempted to pass a routine to a GENERIC routine. Only VAR and CONST parameters can be passed to GENERIC routines.
Programmer Response: If the routine was passed unintentionally, use a different reserved word. If the routine must be passed, rewrite the GENERIC routine as a normal Pascal routine.
System Action: The compiler accepts the routine parameter and continues compiling.

---

170 GENERIC ROUTINES MAY NOT BE PASSED AS PARAMETERS

Explanation: Only VS Pascal routines can be passed as parameters; GENERIC routines cannot.
Programmer Response: One way to work around this limitation is to define a VS Pascal routine that does nothing more than call the GENERIC routine, and then pass the VS Pascal routine in place of the GENERIC routine. Or, if possible, replace the GENERIC directive with the EXTERNAL directive.
System Action: The compiler attempts to recover and continues compiling.

---

171 ONLY STATIC/DEF VARIABLES MAY BE INITIALIZED

Explanation: The only class of variables which may be initialized at compile time are DEF and STATIC variables.
Programmer Response: Check that the variable appearing in the VALUE declaration was intended. If it was, either remove the VALUE declaration or make the variable DEF or STATIC.
System Action: The compiler accepts the VALUE declaration and continues compiling.

172 VARIABLE'S ADDRESS IS NOT COMPILE-TIME COMPUTABLE

Explanation: 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.
Programmer Response: Ensure that any indexes or expressions used on the left side of the VALUE assignment are all constants.
System Action: The compiler attempts to recover and continues compiling.

173 ARRAY STRUCTURE HAS TOO MANY ELEMENTS

Explanation: The indicated array structure contains more elements than was declared for the array type.
Programmer Response: Declare the array type again or use fewer elements in the array constant.
System Action: The compiler starts filling the array from the smallest element and continues compiling.

174 REPETITION FACTOR APPLICABLE TO CONSTANTS ONLY

Explanation: Within an array structure, only a constant expression may be qualified with a repetition factor; a general expression may not.
Programmer Response: Use a compile-time computable value for the value being repeated.
System Action: The compiler attempts to recover and continues compiling.

175 NO CORRESPONDING RECORD FIELD

Explanation: The indicated record structure contains more elements than there are fields within the record type.
Programmer Response: Declare the record type again with more fields or use less fields in the record constant.
System Action: The compiler attempts to recover and continues compiling.

176 THIS IDENTIFIER IS A RESERVED NAME

Explanation: An attempt was made to declare an identifier which is a reserved name.
Programmer Response: Use a new identifier name or check the syntax of your program.
System Action: The compiler attempts to recover and continues compiling.

177 NUMERIC LABELS MUST LIE WITHIN THE RANGE 0..9999.

Explanation: A numeric label declaration was found with a value not in the range 0..9999.
Programmer Response: Use a valid number for the label.
System Action: The compiler accepts the label declaration and continues compiling.

178 IDENTIFIER WAS PREVIOUSLY REFERENCED ILLEGALLY

Explanation: The indicated identifier that was just declared was referenced previously within the associated routine. VS Pascal requires an identifier to be declared *before* its use.

The following would cause such a problem to occur.

```
VAR
  I : INTEGER;
  INTEGER : REAL;
```

Programmer Response: Declare the identifier with a new name.
System Action: The compiler attempts to recover and continues compiling.

179 RECURSIVE REFERENCE WITHIN CONSTANT DECLARATION

Explanation: 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.
Programmer Response: Use a different identifier on one side of the "=."
System Action: The compiler attempts to recover and continues compiling.

180 REPETITION FACTOR NOT APPLICABLE TO RECORD STRUCTURES

Explanation: The indicated record structure contains a component which is qualified with a repetition factor. Only array structures are permitted to have repetition factors.
Programmer Response: Ensure a record constant was intended. If it was, remove the repetition factor and duplicate the field value the correct number of times.
System Action: The compiler ignores the repetition factor and continues compiling.

181 LABEL PREVIOUSLY REFERENCED FROM A GOTO INVALIDLY

Explanation: 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.

Here is an example:

```
BEGIN
  GOTO LABEL1;
  FOR I := 1 TO 10 DO
    BEGIN
      LABEL1: A[I] := 0;
        (* <== Label was previously *)
        (* referenced invalidly.    *)
      .
      .
    END;
END
```

Programmer Response: Correct either the GOTO statement or the label definition.
System Action: The compiler attempts to recover and continues compiling.

182 A GOTO MAY NOT REFERENCE A LABEL WITHIN A SEPARATE STATEMENT SEQUENCE

Explanation: 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.

Here is an example:

```
BEGIN
  FOR I := 1 TO 10 DO
    BEGIN
      LABEL1: A[I] := 0;
      .
      .
    END;
  GOTO LABEL1; (* <== Invalid *)
               (* reference   *)
               (* of label.   *)
END
```

Programmer Response: Correct either the GOTO statement or the label definition.
System Action: The compiler attempts to recover and continues compiling.

183 CASE LABEL OUTSIDE RANGE OF INDEXING EXPRESSION

Explanation: 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
```

Programmer Response: Delete the CASE label or use a larger indexing expression.
System Action: The compiler processes the label and continues compiling. This is a warning.

184 SECOND OPERAND OF MOD OPERATION MUST BE POSITIVE INTEGER

Explanation: The indicated expression involving the MOD operator was found to be invalid; the second operand is required to be a positive integer.
Programmer Response: Use a positive second operand, or define a function to do what is desired.
System Action: The compiler attempts to recover and continues compiling.

185 ROUTINE IS NOT DEFINED IN STANDARD PASCAL

Explanation: This message may be produced when the LANGLVL(ANSI83) compiler option is specified. The indicated call statement refers to a predefined VS Pascal routine which does not exist in standard Pascal.
Programmer Response: If the routine was used intentionally, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler uses the referenced routine and continues compiling This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

186 DIRECTIVE ONLY APPLIES TO PROCEDURE, NOT TO A FUNCTION

Explanation: A function cannot be declared with the MAIN, REENTRANT, or GENERIC routine directives. Only procedures can.
Programmer Response: Remove the directive, or change the function to a procedure which returns the result in a pass-by-VAR parameter.
System Action: The compiler attempts to recover and continues compiling.

187 DIRECTIVE REQUIRES THE PROCEDURE TO HAVE NO PARAMETER LIST

Explanation: A routine declared with the GENERIC routine directive cannot have a formal parameter list.
Programmer Response: Use a different routine directive, or delete the formal parameter list.
System Action: The compiler ignores the formal parameter list and continues compiling.

188 FIRST PARAMETER OF REENTRANT PROCEDURE MUST BE AN INTEGER BY VAR

Explanation: 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 VS Pascal environment is saved between calls.
Programmer Response: Make the first parameter comply with the above, or do not use the REENTRANT directive.
System Action: The compiler attempts to recover and continues compiling.

189 TMP OR BUILTIN PROCEDURE MAY NOT BE PASSED AS A PARAMETER

Explanation: You have used an unsupported routine directive.
Programmer Response: Do not use the routine directive.
System Action: The compiler attempts to recover and continues compiling.

190 TMP PROCEDURE MAY NOT BE INVOKED FROM PASCAL

Explanation: You have used an unsupported routine directive.
Programmer Response: Do not use the routine directive.
System Action: The compiler attempts to recover and continues compiling.

191 SIMPLE CONSTANT REQUIRED

Explanation: A constant expression which required compile-time computation was found where a simple constant is required. This occurs when the RANGE keyword is not used before a constant expression that begins a subrange or when the LANGLVL(ANSI83) compiler option is specified.
Programmer Response: If this occurred in LANGLVL(ANSI83) and was intentional, consider compiling under LANGLVL(EXTENDED). If this occurred under LANGLVL(EXTENDED), use the RANGE keyword before the constant expression.
System Action: The compiler uses the value of the constant expression and continues compiling. This is a warning in LANGLVL(EXTENDED) and is controlled by STDFLAG in LANGLVL(ANSI83).

192 %PERCENT DIRECTIVES ARE NOT RECOGNIZED IN STANDARD PASCAL

Explanation: This message may be produced when the LANGLVL(ANSI83) compiler option is specified. All compiler directives which appear in the source program with the percent (%) prefix are VS Pascal extensions and are not supported in standard Pascal.
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler accepts the directive and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

193 FOR- OR WHILE-LOOP HAS NO STATEMENTS WITHIN ITS BODY

Explanation: 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.

Here are some examples:

```
WHILE A > 0 DO;

FOR I := 1 TO J DO;
```

Programmer Response: Ensure the semicolon after the DO was not placed there by accident.
System Action: The compiler assumes the empty loop was wanted and continues compiling. This is a warning.

194 PACKED SUBRANGES NOT SUPPORTED IN STANDARD PASCAL

Explanation: This message may be produced when the LANGLVL(ANSI83) compiler option is specified. Subrange type definitions may not be PACKED in standard Pascal. This feature is a VS Pascal language extension.
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler packs the subrange and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

195 ACTUAL AND FORMAL PARAMETER SIZES DIFFER

Explanation: The actual parameter being passed by VAR did not have the same storage size as the formal parameter. This could lead to storage overlays, and thus is illegal.
Programmer Response: Ensure the actual and formal parameter requires the same amount of storage or pass the parameter by value or CONST.
System Action: The compiler assumes the types had the same sizes and continues compiling.

198 STANDARD PASCAL FUNCTIONS MAY NOT RETURN STRUCTURED RESULTS

Explanation: Functions in standard Pascal may only return scalar, real, or pointer results.
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler accepts the result type and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

199 NIL NOT ALLOWED IN CONSTANT DEFINITIONS IN STANDARD PASCAL

Explanation: The reserved word NIL is not considered a constant in standard Pascal.
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler sets the constant to NIL and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

200 STANDARD PASCAL DOES NOT SUPPORT CONST PARAMETERS

Explanation: CONST parameters are not allowed in standard Pascal.
Programmer Response: Either use a value or variable parameter or consider compiling in LANGLVL(EXTENDED).
System Action: The compiler generates a pass-by-CONST parameter and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

201 ARRAY ELEMENTS MUST HAVE THE SAME TYPE

Explanation: The array elements of the specified arrays are not identical. This error can occur in LANGLVL(ANSI83) in PACK and UNPACK.
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler accepts the array element type and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

202 FOR INDEX THREATENED BY LOCAL ROUTINE

Explanation: A routine local to the routine being compiled contained a threatening reference to a variable being used as a FOR loop index. Example:

```
PROGRAM BADFOR;
VAR I : INTEGER;
PROCEDURE P;
BEGIN
  I := 3; (* <== This threatens  *)
          (* the FOR loop index. *)
END;
BEGIN
  FOR I := 1 TO 3 DO WRITELN(I);
          (* Error flagged here. *)
END
```

Programmer Response: Save the FOR loop control variable in a temporary variable and use that or use a different variable for the control variable.
System Action: The compiler generates code to modify the FOR loop control variable and continues compiling. The loop may not behave as expected at run time. This is a warning in LANGLVL(EXTENDED) and is controlled by STDFLAG in LANGLVL(ANSI83).

203 NUMBERS MUST BE SEPARATED FROM RESERVED WORDS AND IDENTIFIERS

Explanation: Numeric literals must be separated from reserved words and identifiers by either a comment, a space, or an end-of-line.
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler assumes a space was found and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

204 STRING CONSTANTS MAY NOT BE INDEXED IN STANDARD PASCAL

Explanation: A string constant was being used as a subscripted variable. This is illegal in standard Pascal.
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler generates code to index the string constant and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

205 ALL TAG CONSTANTS NOT SPECIFIED

Explanation: Standard Pascal requires all tag constants that are allowed for a given tag type to be specified as the selector for exactly one variant in a record. Under LANGLVL(ANSI83), this restriction is enforced. Here is an example:

```
RECORD
  CASE BOOLEAN OF
    TRUE: ();
       (* FALSE will not be specified. *)
END    (* Error flagged here.          *)
```

Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler continues compiling. At run time, unpredictable results may occur from attempting to activate a non-existent variant. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

206 STANDARD PASCAL STRINGS MUST HAVE UPPER BOUNDS GREATER THAN ONE

Explanation: A fixed string with an upper bound of one was used in LANGLVL(ANSI83). Fixed strings must have upper bounds greater than one in this language level.
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler uses a one-character fixed string and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

207 STRING TYPES MAY NOT BE READ FROM TEXTFILES IN LANGLVL(ANSI83)

Explanation: Standard Pascal does not allow fixed string types (packed arrays of CHAR) to be read in from files of type TEXT. This is a VS Pascal extension which is flagged in LANGLVL(ANSI83).
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler generates code to read in a fixed string and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

209 PASS BY VAR PARAMETERS MUST HAVE SAME TYPE IN LANGLVL(ANSI83)

Explanation: In standard Pascal, variable parameters must have identical types. This restriction was violated.
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler generates code to pass the parameter by VAR and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

210 TAG FIELDS MAY NOT BE PASSED BY VAR IN LANGLVL(ANSI83)

Explanation: A tag field in a record may not be passed as a variable parameter by either a comment, a space, or an end-of-line.
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler generates code to pass the tag field by VAR and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

211 INPUT WAS NOT INCLUDED IN THE PROGRAM PARAMETER LIST

Explanation: INPUT was not included in the program parameter list, but was used in the program.
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler uses the predefined INPUT file and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

212 OUTPUT WAS NOT INCLUDED IN THE PROGRAM PARAMETER LIST

Explanation: OUTPUT was not included in the program parameter list, but was used in the program.
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler uses the predefined OUTPUT file and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

213 DUPLICATE PROGRAM PARAMETERS NOT ALLOWED

Explanation: An identifier occurred twice in the program parameter list. Duplicate program parameters are not allowed.
Programmer Response: Delete the duplicate program parameter.
System Action: The compiler ignores the duplicate program parameter and continues compiling. This is a warning in LANGLVL(EXTENDED) and is controlled by STDFLAG in LANGLVL(ANSI83).

214 PROGRAM PARAMETERS MUST BE DECLARED AS GLOBAL VARIABLES

Explanation: An identifier listed in the program parameter list was globally declared as something other than a variable. Program parameters (other than INPUT and OUTPUT) must be declared as global variables.
Programmer Response: Remove the program parameter or use a different identifier name.
System Action: The compiler accepts the identifier as declared and continues compiling. This is a warning in LANGLVL(EXTENDED) and is controlled by STDFLAG in LANGLVL(ANSI83).

215 FIXED STRINGS MUST BE OF EQUAL LENGTH IN STANDARD PASCAL

Explanation: Fixed strings used in assignments or binary operations must be of equal length in standard Pascal.
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler continues compiling using the different length fixed strings. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

216 LABELS MUST BE INTEGERS IN STANDARD PASCAL

Explanation: An identifier was used as a label. Labels must be integers in Standard Pascal.
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler attempts to recover and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

217 SET PACKING MUST MATCH IN STANDARD PASCAL

Explanation: In set assignments and binary set operations, both sets must either be packed or unpacked in Standard Pascal.
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler converts one set to an appropriate type and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

218 FUNCTION ASSIGNMENTS MAY NOT BE QUALIFIED IN STANDARD PASCAL

Explanation: Function assignments may be qualified in Standard Pascal.
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler generates code assigning a value into the qualified function result and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

219 FUNCTION RESULTS MAY NOT CONTAIN FILES

Explanation: Because assignments using files are not valid, and every function must have a result assigned to it, function results may not contain files.
Programmer Response: Do not use a function with a result type that contains a file.
System Action: The compiler attempts to recover and continues compiling.

220 PREDEFINED ROUTINES MAY NOT BE PARAMETERS IN STANDARD PASCAL

Explanation: Predefined routines may not be parameters in Standard Pascal.
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: If the routine may be passed as a parameter in LANGLVL(EXTENDED), the compiler generates code to pass the routine and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

221 IDENTIFIER HAS BEEN IMPLICITLY DECLARED BY PROGRAM PARAMETER

Explanation: An identifier has been implicitly declared by a program parameter.
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED). This can occur when declaring INPUT or OUTPUT as a global variable when it was also listed as a program parameter.
System Action: The compiler creates a variable with the name specified and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

222 EMPTY PARAMETER LISTS ARE NOT ALLOWED IN STANDARD PASCAL

Explanation: VS Pascal allows empty parameter lists (a left parenthesis followed by a right parenthesis) on user-defined functions and statements to prevent them from being confused with variables and statements. This is not allowed in Standard Pascal.
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler assumes that no parentheses were used and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

223 FUNCTION CONTAINS NO RESULT ASSIGNMENT

Explanation: A function contains no result assignment.
Programmer Response: Use a procedure or assign a value to the function result.
System Action: The compiler assumes a result was assigned and continues compiling.

224 FORMAL ROUTINE PARAMETER LISTS MUST BE CONGRUENT IN STANDARD PASCAL

Explanation: When passing a routine as an actual parameter, the number of formal parameter sections in the actual routine did not match the number of formal parameter sections in the formal routine.

Consider two routines with the following headers:

```
PROCEDURE P(PROCEDURE Q(I, J : INTEGER));
PROCEDURE R(I : INTEGER; J : INTEGER);
```

Procedure Q has one formal parameter section with two integer parameters, while procedure R has two formal parameter sections each with one integer parameter. Thus, P(R) would result in this message if the unit was not compiled using LANGLVL(EXTENDED).
Programmer Response: If this was intentional, consider compiling with LANGLVL(EXTENDED).
System Action: The compiler assumes the parameter list was congruent and continues compiling. This is only issued in LANGLVL(ANSI83) and its class is controlled by STDFLAG.

225 IDENTIFIER PREVIOUSLY REFERENCED ILLEGALLY IN FORWARD TYPE REFERENCE

Explanation: An identifier was previously referenced illegally in a forward type reference. This can occur as shown below.

```
TYPE
  R = RECORD
        P : @T;
        T : INTEGER; (* T not a type. *)
      END;
```

Programmer Response: Ensure the type and identifier names used were the intended ones. If they were, rename the type or the identifier.
System Action: The compiler attempts to recover and continues compiling.

226 DOUBLE BYTE CHARACTER SET SO HAD NO CORRESPONDING SI

Explanation: The compiler encountered a shift-out (SO), marking the start of a portion of double-byte character set (DBCS) data, but the compiler reached the end of the input line without encountering the shift-in (SI) needed to end the DBCS portion Such a broken SO/SI pair can occur when a single DBCS data portion incorrectly spans multiple lines. This error can occur in DBCS literals. When the GRAPHIC compile-time option is in effect, this error can also occur in string literals, comments, and compiler directives.
Programmer Response: If the indicated character is used intentionally in a string literal, comment, or compiler directive, consider compiling with the NOGRAPHIC compiler option. Otherwise, insert an SI in the proper place.
System Action: The compiler attempts to recover and continues compiling.

227 DOUBLE BYTE CHARACTER SET SI HAD NO CORRESPONDING SO

Explanation: The compiler encountered a shift-in (SI), which marks the end of a portion of double-byte character set (DBCS) data, but the line did not contain the needed shift-out (SO) to mark the beginning of the DBCS data. Such a broken SO/SI pair can occur when a single DBCS portion spans multiple lines, which is not allowed. This error can occur in DBCS literals. When the GRAPHIC compile-time option is in effect, this error can occur in string literals, comments, and compiler directives.
Programmer Response: If the indicated character is used intentionally in a string literal, comment, or compiler directive, consider compiling with the NOGRAPHIC compiler option. Otherwise, insert an SO in the proper place.
System Action: The compiler attempts to recover and continues compiling.

228 INVALID DOUBLE BYTE CHARACTER SET CHARACTER

Explanation: The compiler encountered an invalid double-byte character set (DBCS) character outside the valid DBCS code range. The valid code ranges are '41'X through 'FE'X for the first and second bytes or '4040'X. This error can occur in DBCS literals and hexadecimal literals. When the GRAPHIC compile-time option is in effect, this error can occur in string literals, comments, and compiler directives.
Programmer Response: If you used the indicated character intentionally in a string literal, comment, or compiler directive, consider compiling with the GRAPHIC option. Otherwise, use a valid DBCS character.
System Action: The compiler attempts to recover and continues compiling.

229 DOUBLE BYTE CHARACTER SET BYTE LENGTH IS NOT EVEN

Explanation: The compiler encountered an odd number of bytes of DBCS data between a shift-out (SO) and shift-in (SI). This error can occur in DBCS literals. When the GRAPHIC compile-time option is in effect, this error can also occur in string literals, comments, and compiler directives.
Programmer Response: If you do not intend for the data surrounded by an SO/SI pair to be DBCS data, consider compiling with the NOGRAPHIC compile-time option. Otherwise, correct the byte length of the DBCS data.

230 GLOBAL LABELS MAY ONLY OCCUR IN PROGRAM UNITS

Explanation: A label declaration was found in the global declarations of a SEGMENT unit. Branching to these labels may cause unpredictable results. Only PROGRAM units may have global labels because only programs have code associated with their global declarations and every label must have a definition in the code associated with that declaration level.
Programmer Response: Remove the label declaration. If the label is actually located in the main program, try the following:

```
PROGRAM TEST;          PROGRAM TEST;
LABEL 1;               LABEL 1;
declarations           declarations
BEGIN                  PROCEDURE P1;
code part 1              EXTERNAL;
1: code part 2         PROCEDURE P1;
END.                   BEGIN
                         GOTO 1;
                       END;
                       BEGIN
                       code part 1
                       1: code part 2
                       END.

SEGMENT TEST1;         SEGMENT TEST1;
LABEL 1;               declarations
declarations           PROCEDURE P1;
routine header           EXTERNAL;
BEGIN                  PROCEDURE P;
code part 1            routine header
  GOTO 1;              BEGIN
code part 2            code part 1
END;                     P1;
declarations           code part 2
                       END;
                       declarations
```

System Action: The compiler compiles the label declarations. This is a warning.

231 DOUBLE BYTE CHARACTER SET STRING NOT ENCLOSED IN SHIFT CODES

Explanation: A double-byte character set (DBCS) literal string was not enclosed by shift codes. DBCS data must be enclosed by a starting shift-out (SO) character ('0E'X) and ending shift-in (SI) character ('0F'X).
Programmer Response: If a DBCS string is intended, make sure an SO follows the opening quote and an SI precedes the ending quote. Otherwise, remove the string literal's "G" suffix.
System Action: The compiler assumes shift codes were properly specified and continues compiling.

233 TYPE MUST BE GCHAR OR PACKED ARRAY OF GCHAR

Explanation: The context requires that the indicated expression be of type GCHAR or PACKED ARRAY [1..n] of GCHAR.
Programmer Response: Use the correct type.
System Action: The compiler attempts to recover and continues compiling.

500 RECURSION DETECTED IN "%INCLUDE" PROCESSING *lib(mem)*

Explanation: Source text which was included from member *mem* in library *lib* by means of the %INCLUDE directive contains in itself a %INCLUDE directive which directly or indirectly references the same member recursively. 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.
```

Programmer Response: Ensure the %INCLUDE directives all reference the correct members.
System Action: The compiler halts with a severe error.

---

501 TOO MANY NESTING LEVELS IN "%INCLUDE" PROCESSING *lib(mem)*

Explanation: 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.
Programmer Response: Eliminate the %INCLUDE, or physically put one member within another.
System Action: The compiler halts with a severe error.

---

502 UNABLE TO OPEN "%INCLUDE" LIBRARY: *libname*

Explanation: 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.
Programmer Response: Ensure the library specified was correct.
System Action: The compiler halts with a severe error.

---

503 PERCENT "%" STATEMENT NOT FOUND

Explanation: A "%" symbol was detected, but with no identifier following.
Programmer Response: Delete the percent sign or add the name of the compiler directive desired.
System Action: The compiler attempts to recover and continues compiling.

---

504 PERCENT "%" IDENTIFIER NOT RECOGNIZED

Explanation: An identifier following the "%" symbol is not recognized as a valid compiler directive.
Programmer Response: Use a valid compiler directive name.
System Action: The compiler attempts to recover and continues compiling.

---

505 "ON" OR "OFF" EXPECTED

Explanation: An "ON" or "OFF" was expected but not found.
Programmer Response: Specify "ON" or "OFF" where indicated.
System Action: The compiler attempts to recover and continues compiling.

---

506 UNRECOGNIZABLE OPTION IN "%CHECK"

Explanation: An invalid option was used with a %CHECK directive. Valid options are POINTER, PTR, , SUBSCRIPT, SUBRANGE, FUNCTION, CASE and TRUNCATE.
Programmer Response: Use a valid option or delete the %CHECK directive.
System Action: The compiler attempts to recover and continues compiling.

---

507 %INCLUDE MEMBER NOT FOUND IN LIBRARY

Explanation: The library member which was to be included into the source program could not be found.
Programmer Response: Correct the member name or use the LIB compiler option to specify the library the member is in.
System Action: The %INCLUDE directive is ignored and compilation continues.

---

508 LEFT MARGIN GREATER THAN RIGHT MARGIN

Explanation: The first number (left margin setting) on a %MARGINS compiler directive is greater than the second number (right margin setting).
Programmer Response: Change the numbers on the %MARGINS directive appropriately.
System Action: The compiler uses the previous margin settings and continues compiling.

---

509 MARGINS OVERLAP SEQUENCE FIELD

Explanation: The margin settings on a %MARGINS compiler directive overlap the sequence number field.
Programmer Response: Change the numbers on the %MARGINS directive, or use the SEQUENCE compiler option to change the sequence field placement.
System Action: The compiler assumes that a sequence field is no longer desired and uses the new margin settings. This is a warning.

---

510 MISSING %ENDSELECT DIRECTIVE

Explanation: There is no %ENDSELECT compiler directive to match a previous %SELECT directive. This error can occur when, after encountering an initial %SELECT, the compiler encounters (1) the end of the file, or (2) another %SELECT.
Programmer Response: Insert a %ENDSELECT directive at the appropriate location in the source file, or delete the %SELECT directive.
System Action: If the compiler reached the end of the input file, it stops executing, possibly after issuing other diagnostics. If the compiler found a %SELECT after an initial %SELECT, it ignores the second %SELECT and continues compiling.

---

511 CHARACTERS WERE FOUND BETWEEN %SELECT AND FIRST %WHEN

Explanation: Characters appear after a %SELECT compiler directive but before the first %WHEN directive.
Programmer Response: If the characters must be conditionally compiled, insert a %WHEN directive before them or move them below the appropriate %WHEN directive. Otherwise, move them above the %SELECT directive.
System Action: Characters between %SELECT and the first %WHEN are ignored, and compilation continues. This is a warning.

512 INVALID EXPRESSION IN %WHEN DIRECTIVE
Explanation: A %WHEN compiler directive contains an invalid Boolean expression.
Programmer Response: Correct the syntax of the Boolean expression.
System Action: The compiler continues compiling as if the expression evaluated to FALSE.

513 MISSING %SELECT DIRECTIVE
Explanation: A %WHEN or %ENDSELECT compiler directive appears ahead of a %SELECT directive.
Programmer Response: Insert a %SELECT directive at the appropriate location in the source file, or remove the %WHEN or %ENDSELECT directive.
System Action: The compiler assumes a %SELECT was found and continues compiling.

514 INVALID CONDITIONAL PARAMETER NAME
Explanation: The conditional parameter name on a %WHEN compiler directive is not valid. Note that parameter names used for conditional compilation cannot contain double-byte character set (DBCS) characters or be more than 16 characters long.
Programmer Response: Correct the invalid name.
System Action: The compiler ignores the name and continues compiling.

600 IDENTIFIER *xxxx* IN FORWARD TYPE REFERENCE AT LINE *nnn* OUT OF CONTEXT
Explanation: 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 *)
```

Programmer Response: Make the forward type reference point to the correct type identifier or change the name of the non-type identifier if the type is declared later.
System Action: The compiler attempts to recover and continues compiling.

601 FORWARD TYPE REFERENCE TO *xxxx* AT LINE *nnn* NOT RESOLVED
Explanation: The identifier *xxxx* appeared in a pointer type definition of the form '@*xxxx*' at line *nnn*, but the identifier was not subsequently declared.
Programmer Response: Make the forward type reference point to the correct type identifier or add a new type identifier with the proper name.
System Action: The compiler assumes the type was declared properly and continues compiling.

602 LABEL *xxxx* WAS DECLARED AND/OR REFERENCED BUT WAS NOT DEFINED
Explanation: The label named *xxxx* was declared and/or referenced from within the associated routine, but was not defined. All labels declared in a block must have a corresponding definition in the block's statement part.
Programmer Response: Define the label in this routine or remove the label declaration.
System Action: The compiler assumes the label was defined in this routine and continues compiling. This is a warning in LANGLVL(EXTENDED) and is controlled by STDFLAG in LANGLVL(ANSI83).

603 THE BODY FOR ROUTINE *routinename* WAS NOT DECLARED
Explanation: The routine *routinename* was declared with a directive that requires the routine to have a body, but the routine body was never declared. Directives that require bodies are FORWARD, MAIN, and REENTRANT.
Programmer Response: Change the routine directive to one that does not require a body or declare the routine body.
System Action: The compiler assumes the routine body was resolved and continues compiling.

605 PROGRAM PARAMETER *xxxx* WAS NOT DECLARED AS A GLOBAL VARIABLE
Explanation: The program parameter named *xxxx* in the program heading was not declared in the main program as a global variable.
Programmer Response: Associate the program parameter with a global variable or delete the program parameter.
System Action: The compiler ignores the program parameter and continues compiling. This is a warning in LANGLVL(EXTENDED) and is controlled by STDFLAG in LANGLVL(ANSI83).

700 INPUT FILE SYSIN COULD NOT BE OPENED
Explanation: The input file could not be opened. This could occur because a file was specified which was also used internally by the compiler.
Programmer Response: Specify a new file or rename the file.
System Action: The compiler halts with a severe error.

701 OPTION *option* DOES NOT TAKE SUBOPTIONS
Explanation: *Option* was specified with a suboption list, but it cannot accept suboptions.
Programmer Response: Either (1) remove the suboption list or (2) specify the option for which the suboption list is intended.
System Action: The compiler continues as if no suboption list. This is a warning.

702 *Token* IS NOT A VALID SUBOPTION FOR *option*
Explanation: *Token* is not a valid suboption for compiler option *option*.
Programmer Response: Either do not specify the option at all, or use a valid suboption
System Action: The compiler ignores the option and continues compiling. This is a warning.

703 SUBOPTION EXPECTED FOR *option*
Explanation: *Option* requires a qualifying suboption, but none was found. The left parenthesis indicating a suboption list may have been omitted.
Programmer Response: Either do not specify the option at all, or specify a valid suboption.
System Action: The compiler ignores the option and continues compiling. This is a warning.

704 **MISSING** *element* **EXPECTED FOR** *option*
Explanation: *Option* contained a syntax error because *element* was expected but not found. This can happen if a comma or right parenthesis is missing on a suboption list.
Programmer Response: Either do not specify the option at all or use a syntactically valid form of the option.
System Action: The compiler option is set as specified, and compilation continues. This is a warning.

705 **SUBOPTIONS FOUND WITH NO ASSOCIATED OPTION**
Explanation: A list of suboptions had no option preceding it. This can happen if (1) a left parenthesis immediately follows the left parenthesis that marks the start of an option list, or (2) a left parenthesis immediately follows a suboption list of another option.
Programmer Response: You can (1) add a valid option name before the suboption list, (2) delete the suboption list, or (3) remove the parentheses around the suboption list.
System Action: The compiler ignores the suboption list and continues compiling. This is a warning.

706 **MESSAGE TEXT NOT FOUND FOR LANGUAGE:** *language*
Explanation: Message text does not exist for *language*, as specified on the LANGUAGE option.
Programmer Response: Change the LANGUAGE option to a language that is supported by VS Pascal and implemented on your system.
System Action: Compilation continues; all output will appear in the default language selected during installation. This is a warning.

707 **INVALID CONDPARM PARAMETER NAME:** *name*
Explanation: The conditional parameter name *name* is not valid. Note that conditional parameters cannot contain double-byte character set (DBCS) characters or be more than sixteen characters long.
Programmer Response: Correct the invalid name.
System Action: The compiler ignores the name and continues compiling.

708 **CONDITIONAL PARAMETER** *name* **WAS SET PREVIOUSLY**
Explanation: The conditional parameter *name* is already set.
Programmer Response: Remove the duplicate assignment from the CONDPARM compiler option and restart the compiler.
System Action: The compiler uses the last value specified and continues compiling. This is a warning.

709 *Option1* **AND** *option2* **ARE MUTUALLY EXCLUSIVE;** *option3* **ASSUMED**
Explanation: *Option1* and *option2* cannot be used together.
Programmer Response: Specify non-conflicting options.
System Action: The compiler continues compiling, replacing *option1* with *option3*. This is a warning.

799 **OPTION IS UNSUPPORTED IN VS PASCAL**
Explanation: The indicated compiler option is not supported. Any problems caused by using an option which could cause this message should not be reported to service.
Programmer Response: Remove or correct the specified compiler option if used unintentionally.
System Action: The compiler accepts the option, but the results are unpredictable. This is a warning.

800 **STRING LITERAL CONSTANT IS TOO LONG: EXCEEDS 3190**
Explanation: Because of an implementation restriction, a string constant may not exceed 3190 characters in length.
Programmer Response: Concatenate shorter literals to form a longer string constant.
System Action: The compiler attempts to recover and continues compiling.

801 **ROUTINE NESTING EXCEEDS MAXIMUM**
Explanation: The indicated PROCEDURE or FUNCTION declaration exceeds the maximum allowed nesting level for routines. Routines may be nested to a maximum depth of 8 (including the main program or segment).
Programmer Response: Move the indicated routine a higher scope. This may require moving some identifiers referenced by the routine to a higher scope as well.
System Action: The compiler attempts to recover and continues compiling.

802 **TOO MANY NESTED WITH STATEMENTS OR RECORD DEFINITIONS**
Explanation: This error occurs when too many lexical scopes are active. This can occur in multiple nested WITH statements and RECORD definitions.
Programmer Response: Do not use so many nested WITH statements or RECORD declarations.
System Action: The compiler attempts to recover and continues compiling.

803 **REAL CONSTANT HAS TOO MANY DIGITS**
Explanation: The indicated floating point constant contains more digits than the compiler allows for in scanning. Real constants may have 72 characters in decimal notation, and 16 characters in hexadecimal notation.
Programmer Response: Use fewer digits in the number. Leading zeroes may be replaced with scientific notation, and trailing zeroes may be omitted.
System Action: The compiler attempts to recover and continues compiling.

804 **INTEGER CONSTANT TOO LARGE**
Explanation: The indicated integer constant is not within the range -2147483647 to 2147483647.
Programmer Response: Use an integer in the specified range. If you wish to use the value of -2**31, use the predefined constant MININT.
System Action: The compiler assumes a value of zero and continues compiling.

805 **HEXADECIMAL INTEGER CONSTANT MAY NOT EXCEED 8 DIGITS**
Explanation: The indicated hexadecimal constant exceeds the maximum allowed number of digits.
Programmer Response: Use a smaller constant.
System Action: The compiler attempts to recover and continues compiling.

806 **BINARY INTEGER CONSTANT MAY NOT EXCEED 32 DIGITS**
Explanation: The indicated binary constant exceeds the maximum number of digits.
Programmer Response: Use a smaller constant.
System Action: The compiler assumes a value of zero and continues compiling.

807 ARRAY HAS TOO MANY ELEMENTS
Explanation: The length of the indicated array definition exceeds the addressability of the computer.
Programmer Response: Declare the array with fewer elements or a smaller element type.
System Action: The compiler attempts to recover and continues compiling.

808 MAGNITUDE OF FLOATING POINT CONSTANT TOO LARGE OR TOO SMALL
Explanation: 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:

```
7.23700557733226E75
```

The smallest is:

```
5.39760534693403E-79
```

Programmer Response: Use a valid floating point constant.
System Action: The compiler attempts to recover and continues compiling.

809 MAXIMUM STRING LENGTH EXCEEDED
Explanation: The indicated expression produced a varying length string which exceeds 32767 characters in length. The maximum allowed length for a varying length string is 32767.
Programmer Response: Either truncate the string causing the error or don't use a varying length string.
System Action: The compiler attempts to recover and continues compiling.

810 FIXED POINT OVERFLOW OR DIVIDE-BY-ZERO
Explanation: An integer expression consisting of constant operands causes a program error to occur when it is evaluated.
Programmer Response: Change or rearrange the computation.
System Action: The compiler attempts to recover and continues compiling.

811 THE ORD OF ALL SET MEMBERS MUST LIE WITHIN 0..255
Explanation: The ordinal value of any valid set member may not be less than 0 no greater than 255.
Programmer Response: Modify the set declaration or reference to use appropriate member values.
System Action: The compiler attempts to recover and continues compiling.

812 FLOATING POINT COMPUTATIONAL ERROR
Explanation: The indicated floating point expression causes a program error when evaluated.
Programmer Response: Change or rearrange the computation.
System Action: The compiler continues compiling using an unpredictable result for the expression.

813 DATA STORAGE EXCEEDS ADDRESSABILITY OF MACHINE
Explanation: The storage required to contain all declared variables within a routine or main program exceeds the capacity of the computer; that is, it exceeds 16 megabytes.
Programmer Response: Split the routine up so that it does not use so much storage.
System Action: The compiler attempts to recover and continues compiling.

814 VARIABLE IS NOT PROPERLY ALIGNED
Explanation: 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 fullword integer which has an address that is not on a word boundary would result in this message.)

On most models of the 370 series, the manipulation of objects which are not properly aligned will result in a penalty in execution speed.

This warning will be produced even if the variable is just potential misaligned (as in the case of a subscripted variable).
Programmer Response: Fix the variable alignment if this was unintentional.
System Action: The compiler attempts to recover and continues compiling. This is a warning.

815 OFFSET QUALIFICATION VALUE IS TOO LARGE
Explanation: The indicated field in a record definition is qualified with an offset which would result in a record that was too large to address.
Programmer Response: Use a smaller offset qualification, or use less fields in the record.
System Action: The compiler attempts to recover and continues compiling.

816 OBJECT EXCEEDS STORAGE LIMITS
Explanation: The specified object would cause the program to require more storage than is physically addressable.

Note: This error can occur on a type declaration even if no variable is declared with that type.
Programmer Response: Declare the object so that it won't use so much space.
System Action: The compiler attempts to recover and continues compiling.

817 TOO MANY SCOPES USED
Explanation: The program being compiled had too many scoping levels. Compilation will continue, but some scoping violations will not be detected.
Programmer Response: Use fewer routines, or change some functions to procedures that return results as pass-by-VAR parameters.
System Action: The compiler attempts to recover and continues compiling.

818 TOO MANY %INCLUDE DIRECTIVES USED
Explanation: The number of %INCLUDE directives exceeds the compiler's limit.
Programmer Response: Combine several %INCLUDE members into one member where possible.
System Action: The compiler halts and indicates a severe error.

819 HEX LITERAL HAS INVALID NUMBER OF DIGITS
Explanation: The number of hex digits specified is not a multiple of two in a hex character string (XC) or is not a multiple of four in a hex double-byte character string (XG).
Programmer Response: Make sure the number of hex digits is correct.
System Action: The compiler assumes the literal had the correct number of digits and continues compiling.

900 INTERNAL COMPILER FAILURE

Explanation: An internal compiler error has occurred.

Programmer Response: See *VS Pascal Diagnosis Guide and Reference* for more information.

System Action: The compiler attempts to recover and continues compiling.

AMPL999S INTERNAL COMPILER FAILURE

Explanation: An internal error was detected in the first pass of the compiler. A message will be issued to show the type of internal error detected.

Programmer Response: See *VS Pascal Diagnosis Guide and Reference* for more information.

System Action: Depending on the severity of the message issued after this message, the compiler will continue execution or halt.

Compiler Messages—Intermediate Code Optimization

These messages are issued during the second compiler pass (intermediate code optimization). All messages will indicate the module, routine and statement which caused the message to be issued.

AMPO001S ROUTINE *routinename* IS TOO LARGE TO COMPILE AT STMT *n*

Explanation: 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*.

Programmer Response: Divide the routine into two or more separate routines. For example, you can take this routine

```
PROCEDURE BIG;
  declarations
BEGIN
  code part 1
  code part 2
END;
```

and separate it this way:

```
PROCEDURE BIG;
  declarations
PROCEDURE P1;
  BEGIN
    code part 1
  END;
PROCEDURE P2;
  BEGIN
    code part 2
  END;
BEGIN
  P1;
  P2;
END;
```

System Action: The compiler stops with a return code of 16. No object code is generated.

AMPO002S INTERNAL OPTIMIZER ERROR

Explanation: An internal optimizer error occurred. Another message will identify which routine and statement in the source program caused the error.

Programmer Response: See *VS Pascal Diagnosis Guide and Reference* for further information.

System Action: The compiler generates a trace-back and stops with a return code of 16. No object code is generated.

AMPO700I MESSAGE TEXT NOT FOUND FOR LANGUAGE: *language*

Explanation: Message text does not exist for *language*, as specified on the LANGUAGE option.

Programmer Response: Change the LANGUAGE option to a language that is supported by VS Pascal and implemented on your system.

System Action: Compilation continues; all output will appear in the default language selected during installation.

AMPO999S INTERNAL OPTIMIZER ERROR

Explanation: The compiler detected an internal error during its second pass. A message will be issued to show the type of internal error detected.

Programmer Response: See *VS Pascal Diagnosis Guide and Reference* for further information.

System Action: The compiler stops with a return code of 16. No object code is generated.

Compiler Messages—Object Code Generation

These messages are issued during the third compiler pass (370 object code generation). All messages will indicate the module, routine and statement which caused the message to be issued.

Note: For the AMPT5xx messages, the statement number may be invalid because these messages are issued for declarations.

AMPT001E INEVITABLE NIL POINTER ERROR WILL OCCUR
Explanation: The compiler's code optimizer determined that a NIL pointer checking error will inevitably occur at run time. The message also indicates the module, routine, and statement where error will occur. Here is an example of code that will cause this error:

```
BEGIN
  P := NIL;
  WRITELN(P@.I);
  (*<===AMPT001E - inevitable error*)
END;
```

Programmer Response: Use a valid pointer.
System Action: The compiler attempts to recover and continues compiling.

AMPT002E INEVITABLE HIGH BOUND ERROR WILL OCCUR
Explanation: The compiler's code optimizer determined that a high-bound checking error will inevitably occur at run time. The message also indicates the module, routine, and statement where the error will occur. Here is an example of code that will cause this error:

```
VAR
  I : 1..10;
  J : INTEGER;
BEGIN
  J := 11;
  I := J;
  (*<===AMPT002E - inevitable error*)
END;
```

Programmer Response: Either:

- Adjust the subrange.
- Modify the statement.

System Action: The compiler attempts to recover and continues compiling.

AMPT003E INEVITABLE LOW BOUND ERROR WILL OCCUR
Explanation: The compiler's code optimizer determined that a low-bound checking error will inevitably occur at run time. The message also indicates the module, routine, and statement where the error will occur. Here is an example of code that will cause this error:

```
VAR
  I : 1..10;
  J : INTEGER;
BEGIN
  J := 0;
  I := J;
  (*<===AMPT003E - inevitable error*)
END;
```

Programmer Response: Adjust the subrange or modify the statement.
System Action: The compiler attempts to recover and continues compiling.

AMPT004E INEVITABLE RECORD TAG FIELD ERROR
Explanation: This error occurs when you use a compiler directive not supported by the VS Pascal compiler.

Note: Do not report any problems caused by unsupported features to IBM.
Programmer Response: For best results, remove the unsupported directive from your code.
System Action: The compiler attempts to recover and continues compiling.

AMPT005E FUNCTION ROUTINE DOES NOT RETURN A VALUE
Explanation: The compiler's code optimizer determined that the indicated function does not return a result. Here is an example of code that will cause this error:

```
FUNCTION F(VAR I: INTEGER): INTEGER;
  BEGIN
    READLN(I);
  END;
       (*<===AMPT005 function did not*)
       (*return a result*)
```

Programmer Response: Correct the function so that it returns a result.
System Action: The compiler attempts to recover and continues compiling.

AMPT500W RECORD TYPE CONTAINS TOO MANY FIELDS (MAX = *n*)
Explanation: The module being compiled with the DEBUG option contains a record type definition with too many fields to be accommodated in the Interactive Debugging Tool's type table. The maximum number of table entries possible in your implementation of VS Pascal is *n*.
Programmer Response: You can ignore the error, but the resulting code might not work properly with the Interactive Debugging Tool. Otherwise, compile the program without the DEBUG option.
System Action: The compiler attempts to recover and continues compiling.

AMPT501W FIELD NAME SPACE POOL OVERFLOWED
Explanation: The module being compiled with the DEBUG option contains a large number of record definitions. The Interactive Debugging Tool table that contains record field names overflowed.
Programmer Response: You can ignore the error, but the resulting code might not work properly with the Interactive Debugging Tool. Otherwise, compile the program without the DEBUG option.
System Action: The compiler attempts to recover and continues compiling.

AMPT502E TYPE TABLE OVERFLOW. DEBUG IS DISABLED
Explanation: The module being compiled with the DEBUG option contains more than 8,192 unique data types, exceeding the capacity of the Interactive Debugging Tool's type table. The Interactive Debugging Tool cannot be used on this module.
Programmer Response: Either:

- Ignore the error.
- Do not compile with the DEBUG option.

System Action: The compiler attempts to recover and continues compiling.

AMPT503W SYMBOL NAME SPACE POOL OVERFLOWED
Explanation: The module being compiled with the debug option contains a large number of symbols. The Interactive Debugging Tool table that contains symbol names overflowed.
Programmer Response: You can ignore the error, but the resulting code might not work properly with the Interactive Debugging Tool. Otherwise, compile the program without the DEBUG option.
System Action: The compiler attempts to recover and continues compiling.

AMPT700I MESSAGE TEXT NOT FOUND FOR LANGUAGE: *language*
Explanation: You specified *language* on the LANGUAGE option, but no message text exists for that language identifier.
Programmer Response: Specify a LANGUAGE identifier that is supported by VS Pascal and implemented on your system.
System Action: Compilation continues; all output appears in the default language selected during installation.

AMPT900S EXPRESSION IS TOO COMPLICATED
Explanation: The indicated expression is too complex to compile. This error typically occurs when a statement requires too many registers.
Programmer Response: Break the statement up into multiple statements. If the indicated statement contains a relatively simple expression, see the *VS Pascal Diagnosis Guide and Reference* for more help.
System Action: The compiler stops compiling the program.

AMPT901S ROUTINE *routinename* **CONTAINS TOO MANY STATEMENTS (MAX =** *n***)**
Explanation: The table that contains statement information overflowed in the indicated routine *routinename*. The maximum number of statements you can use is *n*.
Programmer Response: Divide the routine into two or more routines. For example, you can take this routine:

```
PROCEDURE BIG;
  declarations
BEGIN
  code part 1
  code part 2
END;
```

and separate it this way:

```
PROCEDURE BIG;
  declarations
PROCEDURE P1;
  BEGIN
    code part 1
  END;
PROCEDURE P2;
  BEGIN
    code part 2
  END;
BEGIN
  P1;
  P2;
END;
```

System Action: The compiler stops compiling the program.

AMPT902S ROUTINE EXCEEDS 8K LIMIT
Explanation: A routine generated more than 8192 bytes of code. Because VS Pascal reserves only two base registers to address code, 8192 bytes is the limit. The routine causing the problem and the statement where the limit was exceeded are shown following the message. This error also occurs with the DEBUG option in effect when the information required by the Interactive Debugging Tool exceeds 8192 bytes of storage.
Programmer Response: Divide the routine into two or more routines. For example, you can take this routine:

```
PROCEDURE BIG;
  declarations
BEGIN
  code part 1
  code part 2
END;
```

and separate it this way:

```
PROCEDURE BIG;
  declarations
PROCEDURE P1;
  BEGIN
    code part 1
  END;
PROCEDURE P2;
  BEGIN
    code part 2
  END;
BEGIN
  P1;
  P2;
END;
```

System Action: The compiler stops.

AMPT998S INTERNAL TRANSLATOR ERROR
Explanation: An internal translator error occurred. A later message will identify which routine and statement caused the error.
Programmer Response: See *VS Pascal Diagnosis Guide and Reference* for more information.
System Action: The compiler generates a trace-back and stops. No object code is generated.

AMPT999S INTERNAL TRANSLATOR ERROR
Explanation: The compiler detected an internal error on its third pass. Another message will indicate the type of error detected.
Programmer Response: See *VS Pascal Diagnosis Guide and Reference* for more information.
System Action: Depending on the severity of the message issued after this message, the compiler continues running or stops.

Run-Time Messages

These messages are issued during execution of a program. The message severity code letter for each message is a default and may be changed by the ONERROR procedure. See Appendix C, "Run-Time Error Default Actions" on page 267 for a list of the default actions for run-time error messages passed to ONERROR in FACTION. The action associated with each message class is shown below:

Message Class	Return Code	Action
Informational	0	None
Error	8	The error count is decremented
Severe error	16	Program execution halts

AMPX011S OPERATION EXCEPTION
Explanation: An operation exception occurred in the program. This error is typically caused by either:
- An assembly language routine linked with your Pascal program
- A "wild" assignment through an uninitialized pointer.

Programmer Response: Correct the error.
System Action: The program ends.

AMPX012S PRIVILEGED EXCEPTION
Explanation: A privileged exception occurred in the program. This error is typically caused by an assembly language routine linked with your Pascal program.
Programmer Response: Correct the error.
System Action: The program ends.

AMPX013S EXECUTE EXCEPTION
Explanation: An execute exception occurred in the program. This error is typically caused by an assembly language routine linked with your Pascal program.
Programmer Response: Correct the error.
System Action: The program ends.

AMPX014S PROTECTION EXCEPTION
Explanation: A protection exception occurred in the program. This error is typically caused by either:
- A "wild" assignment through an uninitialized pointer
- An array assignment with a bad subscript (with checking off).

Programmer Response: Correct the error.
System Action: The program ends.

AMPX015S ADDRESSING EXCEPTION
Explanation: An addressing exception occurred in the program. This error is typically caused by either:
- A "wild" assignment through an uninitialized pointer
- An array assignment with a bad subscript (with checking off).

Programmer Response: Correct the error.
System Action: The program ends.

AMPX016S SPECIFICATION EXCEPTION
Explanation: A specification exception occurred in the program. This error is typically caused by an assembly language routine linked with your Pascal program.
Programmer Response: Correct the error.
System Action: The program ends.

AMPX017S DATA EXCEPTION
Explanation: A data exception occurred in the program. This error is typically caused by a non-Pascal routine linked with a Pascal program.
Programmer Response: Correct the error.
System Action: The program ends.

AMPX018E FIXED POINT OVERFLOW EXCEPTION
Explanation: A fixed-point overflow exception occurred in the program. This error is typically caused when an addition, subtraction, or multiplication produces an integer greater than MAXINT.
Programmer Response: Correct the error.
System Action: The run-time error counter is decreased. The program continues running.

AMPX019E FIXED POINT DIVIDE BY ZERO EXCEPTION
Explanation: A fixed-point, divide-by-zero exception occurred in the program. This error occurs when the second operand (the divisor) of a DIV or MOD operation has a value of zero.
Programmer Response: Correct the error.
System Action: The run-time error counter is decreased. The program continues running.

AMPX020E DECIMAL OVERFLOW EXCEPTION
Explanation: A decimal overflow exception occurred in the program. This error typically occurs in a non-Pascal routine linked to the Pascal program.
Programmer Response: Correct the error.
System Action: The run-time error counter is decreased. The program continues running.

AMPX021E DECIMAL DIVIDE BY ZERO EXCEPTION
Explanation: A decimal divide-by-zero exception occurred in the program. This error typically occurs in a non-Pascal routine linked to the Pascal program.
Programmer Response: Correct the error.
System Action: The run-time error counter is decreased. The program continues running.

AMPX022E EXPONENT OVERFLOW EXCEPTION
Explanation: An exponent overflow exception occurred in the program. This error typically occurs when the result of a floating-point multiplication or division is greater than 7.23700557733226E75.
Programmer Response: Correct the error.
System Action: The run-time error counter is decreased. The program continues running.

AMPX023E EXPONENT UNDERFLOW EXCEPTION
Explanation: An exponent underflow exception occurred in the program. This error typically occurs when the result of a floating-point multiplication or division is less than 5.39760534693403E-79.
Programmer Response: Correct the error.
System Action: The run-time error counter is decreased. The program continues running.

AMPX024E SIGNIFICANCE EXCEPTION
Explanation: The VS Pascal run-time environment does not intercept this exception. This error is typically caused by modifying the VS Pascal run-time environment.
Programmer Response: If the VS Pascal run-time environment was not modified locally, contact your local system support.
System Action: The run-time error counter is decreased. The program continues running.

AMPX025E FLOATING POINT DIVIDE BY ZERO EXCEPTION
Explanation: A floating-point, divide-by-zero exception occurred in the program. This error is caused by an attempt to divide by zero.
Programmer Response: Correct the error.
System Action: The run-time error counter is decreased. The program continues running.

AMPX026S SEGMENT TRANSLATION EXCEPTION
Explanation: This is a non-Pascal system error.
Programmer Response: Run the program again and, if the error persists, consult the *VS Pascal Diagnosis Guide and Reference* for more help.
System Action: The program ends.

AMPX027S PAGE TRANSLATION EXCEPTION
Explanation: This is a non-Pascal system error.
Programmer Response: Run the program again and, if the error persists, consult the *VS Pascal Diagnosis Guide and Reference* for more help.
System Action: The program ends.

AMPX028S TRANSLATION SPECIFICATION EXCEPTION
Explanation: This is a non-Pascal system error.
Programmer Response: Run the program again and, if the error persists, consult the *VS Pascal Diagnosis Guide and Reference* for more help.
System Action: The program ends.

AMPX029S SPECIAL OPERATION EXCEPTION
Explanation: This is a non-Pascal system error.
Programmer Response: Run the program again and, if the error persists, consult the *VS Pascal Diagnosis Guide and Reference* for more help.
System Action: The program ends.

AMPX030E TERMINAL ATTENTION EXCEPTION
Explanation: An attention was signaled from the user's terminal.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX031E LOW BOUND CHECKING ERROR
Explanation: Either:
- The value of an array subscript is less than the minimum allowed for the subscript.
- The value being assigned to a subrange type variable is less than the minimum allowed for the subrange.

This error can also occur when the second operand (the divisor) of a MOD operation is less than or equal to zero.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX032E HIGH BOUND CHECKING ERROR
Explanation: Either:
- The value of an array subscript is greater than the maximum allowed for the subscript.
- The value being assigned to a subrange type variable is greater than the maximum allowed for the subrange.

Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX033E NIL POINTER CHECKING ERROR
Explanation: An attempt was made to reference a dynamic variable from a pointer that has the value NIL.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX034E CASE LABEL CHECKING ERROR
Explanation: The expression of a CASE statement has a value other than any of the specified CASE labels and there is no OTHERWISE clause.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX035E FUNCTION VALUE CHECKING ERROR
Explanation: A function routine returned to its invoker without being assigned a result.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX036E ASSERTION FAILURE CHECKING ERROR
Explanation: The expression of an ASSERT statement computed to the value FALSE.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX037E STRING SUBSCRIPT OUT OF BOUNDS CHECKING ERROR
Explanation: The subscript on a STRING is not in the range 0..LENGTH(*s*), where *s* is the STRING being subscripted.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX039E STRING TRUNCATION CHECKING ERROR
Explanation: The program contains a string assignment in which the length of the source string was longer than the maximum length of the target string. VS Pascal does not implicitly truncate strings in such a case.
Programmer Response: Correct the error.
System Action: VS Pascal makes the assignment, possibly overlaying storage used by other variables. The run-time error counter decreases. The program continues running.

AMPX040E STRING MAXIMUM LENGTH EXCEEDED
Explanation: Concatenation caused a STRING or GSTRING to exceed 32,767 characters.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX041S FILE COULD NOT BE OPENED: *ddname*
Explanation: An error occurred when the program attempted to open the file named *ddname*. The most probable cause of this error is a missing DD statement. Under CMS, this error can occur when the program tries to open a file with a format other than "F" or "V".
Programmer Response: Correct the error.
System Action: The program ends.

AMPX042E LRECL SIZE TOO SMALL FOR FILE *ddname*
Explanation: The logical record length of the file named *ddname* is not large enough to contain a single file component.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX043E FILE IS NOT OPEN FOR OUTPUT: *ddname*
Explanation: The program attempted an output operation on the file named *ddname*, but the file is open for input.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX044E FILE IS NOT OPEN FOR INPUT: *ddname*
Explanation: The program attempted an input operation on the file named *ddname*, but the file is open for output.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX045E LOGICAL RECORD IS TOO SMALL IN INPUT FILE
Explanation: The program is reading a record file that consists of variable-length records (RECFM = V). However, the program read a logical record that is too short to represent a valid record in the file.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX046E DATA LARGER THAN LRECL FOR FILE
Explanation: The logical record length of a file is too small to contain the file's component.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX048E MISSING MEMBER IN FILE: *member library*
Explanation: The indicated *member* cannot be found in the partitioned data set.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX049E FLOATING POINT UNDERFLOW OR OVERFLOW
Explanation: The floating-point number read by the READ procedure was either too small or too large to be represented within the machine.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX050E BLKSIZE EXCEEDS 32760 IN FILE: *ddname*
Explanation: The block size for the file named *ddname* exceeds the maximum of 32,760 bytes.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX051E LRECL > BLKSIZE-4 IN V FORMAT FILE: *ddname*
Explanation: The logical record length of the file named *ddname* is too large to permit at least one record to be fit in a block.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX052E BLKSIZE NOT INTEGER MULTIPLE OF LRECL IN: *ddname*
Explanation: The block size of the fixed-length record file named *ddname* is not an integer multiple of the logical record length.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX053E COMPONENT LENGTH OF FILE EXCEEDS 32760 IN: *ddname*
Explanation: The component length of the file named *ddname* exceeds the 32,760-byte maximum. Because a single element must fit in one logical record, its length cannot exceed 32,760 bytes.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX054E GET OR READ CALLED AFTER END-OF-FILE IN: *ddname*
Explanation: The program tried to advance the file named *ddname* beyond the end of the file.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX055E INTEGER READ OPERATION FAILED FOR FILE: *ddname*

Explanation: The program tried to read an integer from the text file named *ddname*, but either:

- The end-of-file occurred.
- An unrecognizable character appeared where the integer should have been.

Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX056E OVERFLOW/UNDERFLOW DETECTED IN INTEGER READ: *ddname*

Explanation: In the file named *ddname*, an attempt was made to read an integer whose value lies outside the permissible range of -2147483648..2147483647.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX058I OPEN AND INTERACTIVE ARE NO LONGER SUPPORTED; USE RESET OR REWRITE

Explanation: The procedures OPEN and INTERACTIVE are not supported. *VS Pascal Application Programming Guide* and *VS Pascal Language Reference* describe the equivalent operations.
Programmer Response: Correct the error.
System Action: The program continues running.

AMPX059E TEXT EXCEEDS LOGICAL RECORD LENGTH IN FILE: *ddname*

Explanation: The line of data being written into the text file named *ddname* exceeds the file's logical record length. VS Pascal ends the line at end of the logical record and places the remaining text of the line in the next logical record.

VS Pascal issues this message on the first occurrence for each file; it does not issue the message for subsequent occurrences in the same file.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX060E OPERAND TO RELEASE DOES NOT CORRESPOND TO MARK

Explanation: The parameter passed to RELEASE does not have the value returned by a call to MARK.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX061E OPERAND TO DISPOSE NOT ALLOCATED WITH NEW

Explanation: The program contains a DISPOSE operation for a pointer whose value was not returned by NEW.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX062E REAL READ OPERATION FAILED FOR FILE *ddname*

Explanation: The program tried to read a REAL data type from a text file, but either:

- The end-of-file occurred.
- An unrecognizable character appeared where the REAL should have been.

Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX063E OPERAND TO DISPOSE ALREADY DEALLOCATED

Explanation: A pointer referred to on a DISPOSE operation points to heap storage that has already been released.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX064E INSUFFICIENT SPACE TO DO NEW

Explanation: There is not enough available storage to perform the NEW procedure.

Under MVS, run the program in a larger region.

Under CMS, run the program in a larger virtual machine.

You cannot call DISPOSE for storage you no longer need.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX065E STORAGE HAS BEEN INCORRECTLY ASSIGNED PRIOR TO DISPOSE

Explanation: The program uses the pointer being disposed of incorrectly. Specifically, the pointer causes the heap to be modified beyond the size of the dynamic variable. This can happen when the dynamic variable is a RECORD allocated by specifying tag values, and the variable is later used in an assignment with a different variant.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX066E OPERAND TO DISPOSE IS NIL OR UNDEFINED.

Explanation: The operand is not valid for DISPOSE.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX067E HEAP INCORRECT DUE TO PREVIOUS INVALID ASSIGNMENT USING A POINTER

Explanation: The heap has been damaged. The damage was probably occurred when a pointer was used incorrectly.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX068S INTERNAL RUN-TIME ERROR

Explanation: This is a run-time environment error.
Programmer Response: See *VS Pascal Diagnosis Guide and Reference* for more information.
System Action: The program ends.

AMPX069S INTERNAL RUN-TIME ERROR
Explanation: This is a run-time environment error.
Programmer Response: See *VS Pascal Diagnosis Guide and Reference* for more information.
System Action: The program ends.

AMPX070E LN: ARGUMENT < = 0.0
Explanation: The natural logarithm function (LN) was called with a 0 or negative argument.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX071E SQRT: ARGUMENT < 0.0, ZERO RETURNED AS RESULT
Explanation: The square root function (SQRT) was called with a negative argument.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX072E EXP: ARGUMENT TOO LARGE, EXCEEDS 174.67309
Explanation: 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.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX073E RANDOM: SEED IS OUT OF RANGE
Explanation: The argument of the RANDOM function is either:
- Negative
- Greater than the maximum of 1048575.

Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX074E SIN/COS: ARGUMENT TOO LARGE, EXCEEDS (PI/2)50**
Explanation: The argument in a call to SIN or COS is too large for an accurate result to be computed.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX075E SEEK CALLED FOR A FILE NOT OPENED FOR DIRECT ACCESS
Explanation: A call to SEEK refers to a file that is not open for direct-access processing.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX076E SEEK: BAD RELATIVE RECORD ADDRESS
Explanation: The record number in an invocation of SEEK has an invalid value.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX077E DIRECT ACCESS FILE DOES NOT HAVE FIXED UNBLOCKED RECORDS: *ddname*
Explanation: The program tried to perform direct-access (relative record) operations on a file that is either not fixed or not unblocked. The required record format for a file to be manipulated with SEEK is RECFM=F.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX078E TARGET STRING FILLED TO MAXIMUM LENGTH IN WRITESTR CALL
Explanation: The target STRING (first parameter) in a call to WRITESTR was filled to capacity before the data being assigned into the STRING was exhausted.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX079E SOURCE STRING EXHAUSTED IN READSTR CALL
Explanation: Before reading all data from the source string (first parameter), the end of the string was encountered.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX080E LTOKEN: TOKEN LENGTH EXCEEDS MAXIMUM LENGTH OF STRING
Explanation: The token to be returned on a call to LTOKEN is longer than the maximum length of the string passed to receive it.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX081E *functionname*: **PADDING EXCEEDS MAXIMUM LENGTH OF STRING**
Explanation: The function *functionname* specifies a pad length (second operand) that exceeds the maximum allowed length of the target string (first parameter).
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX082E *functionname*: **LENGTH PARAMETER LESS THAN ZERO**
Explanation: The function *functionname* specifies a length parameter less than zero.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX083E *functionname*: **STARTING INDEX IS LESS THAN 1**
Explanation: The function *functionname* specifies a starting index less than one.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX084E *functionname*: **SUBSTRING NOT CONTAINED WITHIN SOURCE STRING**
Explanation: The function *functionname* specifies a substring not contained in the source string.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX085E **SET OPERATION OUT OF BOUNDS**
Explanation: The set resulting from a set operation contains members that are outside the range of the target set. This can occur in a set assignment in which the source set contains members that are not valid for the declared type of the target set.
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX090E **PICTURE: CHARACTERS "Z", "*", AND DRIFTING SIGN CAN NOT BE MIXED**
Explanation: Each field (decimal and exponent) can contain only one drifting character.
Programmer Response: Correct the error in the PICTURE specification string.
System Action: The PICTURE function returns a null string.
System Action: The run-time error counter decreases. The program continues running.

AMPX091E **PICTURE: INVALID CHARACTER IN DECIMAL FIELD:** *character*
Explanation: A decimal field cannot contain *character*.
Programmer Response: Correct the error in the PICTURE specification string.
System Action: The PICTURE function returns a null string.
System Action: The run-time error counter decreases. The program continues running.

AMPX092E **PICTURE: INVALID CHARACTER IN PICTURE SPECIFICATION:** *character*
Explanation: *Character* is not a valid PICTURE character. A syntax error in the PICTURE specification can also produce this message.
Programmer Response: Correct the error in the PICTURE specification string.
System Action: The PICTURE function returns a null string.
System Action: The run-time error counter decreases. The program continues running.

AMPX093E *functionname*: **DOUBLE BYTE CHARACTER SET SO HAD NO CORRESPONDING SI**
Explanation: During a call to the function *functionname*, VS Pascal encountered a starting shift-out (SO) but no corresponding shift-in (SI). Double-byte character set (DBCS) data must be enclosed by a starting SO character ('0E'X) and ending SI ('0F'X).
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX094E *functionname*: **DOUBLE BYTE CHARACTER SET SI HAD NO CORRESPONDING SO**
Explanation: During a call to the function *functionname*, VS Pascal encountered a shift-in (SI) before a shift-out (SO). Double-byte character set (DBCS) data must be enclosed by a starting SO character ('0E'X) and ending SI ('0F'X).
Programmer Response: Correct the error.
System Action: The run-time error counter decreases. The program continues running.

AMPX095E *functionname*: **INVALID DOUBLE BYTE CHARACTER SET CODE RANGE**
Explanation: A double-byte character set (DBCS) string or the DBCS portion of a mixed string is invalid because of an improper code range. Proper code ranges are either:
- '41'X through 'FE'X for the first and second bytes
- '4040'X.

VS Pascal encountered this error during a call to the function *functionname*.
Programmer Response: Use only valid DBCS characters in the proper range.
System Action: The run-time error counter decreases. The program continues running.

AMPX096E *functionname*: **DOUBLE BYTE CHARACTER SET BYTE LENGTH IS NOT EVEN**
Explanation: Either:
- The byte length of a double-byte character set (DBCS) string is not even.
- The byte length of the DBCS portion of a mixed string is not even.

VS Pascal encountered this error during a call to the function *functionname*.
Programmer Response: Change the string so that the byte length is even.
System Action: The run-time error counter decreases. The program continues running.

AMPX097E *functionname*: **STRING CONTAINS SINGLE BYTE CHARACTER SET CHARACTERS**
Explanation: A string that must contain only double-byte character set (DBCS) characters contains single-byte character set (SBCS) characters. VS Pascal encountered this error during a call to the function *functionname*.
Programmer Response: Change the string so that it contains only valid DBCS characters.
System Action: The run-time error counter decreases. The program continues running.

AMPX098E **DBCS READ OPERATION FAILED FOR FILE:** *ddname*
Explanation: While reading double-byte character set (DBCS) data into the GCHAR or GSTRING variable named *ddname*, VS Pascal encountered single-byte character set (SBCS) data. This can happen when the field width specified for the input data mistakenly overlaps into SBCS characters.
Programmer Response: Make sure the data to be read are valid DBCS characters, or use a shorter (or no) field width for the input data.
System Action: The run-time error counter decreases. The program continues running.

AMPX099E SEGMENTS MAY NOT BE CALLED AS ROUTINES
Explanation: The program called a segment as if it were a routine. Therefore, you have duplicate names for:
- An external routine in another unit
- A segment in the program.

Programmer Response: Either:
- In the segment, include an entry point with the same name as the segment.
- Declare the external routine with another name.

System Action: The run-time error counter decreases. The program continues running.

AMPX100E INSUFFICIENT SPACE TO DO NEWHEAP
Explanation: There is not enough available storage to perform the NEWHEAP procedure.
Programmer Response: Use DISPOSEHEAP to deallocate unused heaps. Otherwise:

Under MVS, run the program in a larger region.

Under CMS, run the program in a larger virtual machine.

System Action: The run-time error counter decreases. The program continues running.

AMPX101E OPERAND TO DISPOSEHEAP IS NOT A VALID HEAP
Explanation: The pointer variable passed to the DISPOSEHEAP procedure does not identify an existing heap.
Programmer Response: Either:
- Pass a heap identifier returned by NEWHEAP.
- Call DISPOSE or RELEASE for this pointer as appropriate.

System Action: The run-time error counter decreases. The program continues running.

AMPX102E OPERAND TO USEHEAP IS NOT A VALID HEAP
Explanation: The pointer variable passed to the USEHEAP procedure does not identify an existing heap.
Programmer Response: Pass a heap identifier returned by NEWHEAP.
System Action: The run-time error counter decreases. The program continues running.

AMPX200I THE MODULE MUST BE LINKED WITH DEBUG FOR DEBUGGER FEATURES
Explanation: You can use the Interactive Debugging Tool only on modules that were linked-edited with the DEBUG option.
Programmer Response: If you want to use the Interactive Debugging Tool, link-edit the module with the DEBUG option. Otherwise, do not use the DEBUG run-time option.
System Action: The program continues running. The ONERROR routine cannot trap this error.

AMPX201I THE MODULE MUST BE COMPILED WITH DEBUG FOR SYMBOLIC DUMP
Explanation: A run-time error occurred, and a symbolic dump of the offending routine was attempted. However, the module in which the routine is located was not compiled with the DEBUG option.
Programmer Response: If you want a symbolic dump, compile the module with the DEBUG option and link-edit the module with the DEBUG option.
System Action: The program continues running. The ONERROR routine cannot trap this error.

AMPX203E ERROR OCCURRED WHILE EXECUTING ONERROR ROUTINE
Explanation: A run-time error occurred while the ONERROR routine was running. ONERROR is a procedure that you write to diagnose run-time errors and specify a course of action.
Programmer Response: Correct the error.
System Action: The program continues running. The ONERROR routine cannot trap this error.

AMPX204S NO VS PASCAL ENVIRONMENT FOUND FOR SUBPROGRAM DIRECTIVE
Explanation: An unsupported directive was used, and no run-time environment could be found.
Programmer Response: Use the MAIN directive.
System Action: The program ends. The ONERROR routine cannot trap this error.

AMPX600E INVALID INPUT/OUTPUT OPTION: *option*
Explanation: The option string passed to the procedure contains the incorrect or invalid option *option*.
Programmer Response: Correct or change *option*.
System Action: The run-time error counter decreases. The program continues running.

AMPX601E INVALID VALUE *suboption* **FOR INPUT/OUTPUT OPTION** *option*
Explanation: The invalid value *suboption* was assigned to the input/output option *option*.
Programmer Response: Assign a valid value to *option*.
System Action: The run-time error counter decreases. The program continues running.

AMPX602E INVALID SYNTAX IN INPUT/OUTPUT OPTIONS: *option*
Explanation: The options string contains a syntax error in or before *option*. This error can be caused by a missing equal sign, comma, or option value.
Programmer Response: Correct the syntax of the options.
System Action: The run-time error counter decreases. The program continues running.

AMPX603E INVALID NEWHEAP OPTION: *option*
Explanation: The option string passed to the NEWHEAP procedure contains the invalid option *option*.
Programmer Response: Either:
- Replace *option* with a valid option.
- Delete *option* altogether.

System Action: The run-time error counter decreases. The program ignores the invalid option and continues running.

AMPX604E INVALID VALUE *suboption* **FOR NEWHEAP OPTION** *option*
Explanation: In the option string passed to the NEWHEAP procedure, the invalid value *suboption* was assigned to *option*.
Programmer Response: Either:
- Replace the incorrect *suboption* with a valid value.
- Delete *option* from the NEWHEAP option string.

System Action: The run-time error counter decreases. The program uses the default value for *option* and continues running.

AMPX605E INVALID SYNTAX IN NEWHEAP OPTIONS: *option*
Explanation: The option string passed to the NEWHEAP procedure contained a syntax error in or before *option*. This error is caused by:

- Missing commas between the options
- Missing equal signs between an option and its value
- A missing value for the option.

Programmer Response: Fix the syntax error.
System Action: The run-time error counter decreases. The program ignores the invalid option and continues running.

AMPX700I INVALID RUN-TIME OPTION: *option*
Explanation: You invoked a VS Pascal program with the invalid run-time option *option*. When invoking a program, you must precede run-time options with a slash, "/".
Programmer Response: If the parameters you pass to the program contain a slash, you must still precede those parameters with a slash. Otherwise, either:

- Use a valid run-time option.
- Fix the syntax of the options.

System Action: The program continues running.

AMPX701I INVALID VALUE *suboption* **SPECIFIED WITH RUN-TIME OPTION** *option*
Explanation: *Suboption* is not a valid attribute for run-time option *option*. VS Pascal also issues this message when a numeric suboption is outside its valid range.
Programmer Response: Either:

- Specify a valid attribute for *option*.
- Use another option.

System Action: The program continues running.

AMPX702I INVALID SYNTAX IN RUN-TIME OPTION: *option*
Explanation: The run-time option *option* contains a syntax error. The error might be a missing attribute or an incorrectly specified attribute. You must precede attributes for run-time options with a left parenthesis, "(", and end them with a right parenthesis, ")".
Programmer Response: Correct the syntax of the option.
System Action: The program continues running.

AMPX703I MESSAGE TEXT NOT FOUND FOR LANGUAGE: *language*
Explanation: Message text does not exist for *language*, as specified on the LANGUAGE option.
Programmer Response: Change the LANGUAGE option to a language that is supported by VS Pascal and implemented on your system.
System Action: The program continues running. All output will appear in the default language selected during installation.

AMPX900S EXECUTION NOT ALLOWED TO CONTINUE
Explanation: An error was issued that had XHALT returned in FACTION after calling the ONERROR procedure.
Programmer Response: If you want the program to continue, write an ONERROR procedure that removes XHALT from FACTION for this message number.
System Action: The program ends. The ONERROR routine cannot trap this error.

AMPX901S PROGRAM HALTED DUE TO ERROR COUNT
Explanation: The number of errors you specified on the ERRCOUNT run-time option (or its default value) was reached.
Programmer Response: If you want the program to continue, either:

- Specify a larger ERRCOUNT value.
- Write an ONERROR procedure that removes XDECERR from FACTION for some messages.

System Action: The program ends. The ONERROR routine cannot trap this error.

AMPX902S PROGRAM INTERRUPT OCCURRED IN NON-PASCAL ROUTINE
Explanation: A program interrupt occurred in a non-Pascal routine. This message is issued because certain registers or data areas contain values different from those the VS Pascal run-time environment expects. If this error was caused by an abend, the abend code will be in register 2 and the abend address will be in register 3.
Programmer Response: Fix the non-Pascal routine. If no non-Pascal routine was called, consult *VS Pascal Diagnosis Guide and Reference* for more help.
System Action: The program ends. The ONERROR routine cannot trap this error.

AMPX999S RECURSIVE ERROR IN RUN-TIME ENVIRONMENT
Explanation: A second error occurred while VS Pascal was recovering from the first error. This error can be caused by "wild" pointer assignments.
Programmer Response: Ensure that no invalid pointer assignments have been made. If this error persists, consult *VS Pascal Diagnosis Guide and Reference* for more help.
System Action: The program ends.

Interactive Debugging Tool Messages

These messages are issued while in the Interactive Debugging Tool. They are all informational messages.

AMPD500I CURRENT MODULE NOT COMPILED WITH DEBUG OPTION
Explanation: The current module contains a debugging command, but the module was not compiled with the DEBUG option.
Programmer Response: Either:
- Do not execute the debugging command in this module.
- Recompile the module with the DEBUG option.

System Action: VS Pascal ignores the command.

AMPD501I NO STATEMENT *nnn* IN *routinename*
Explanation: A BREAK command specifies a statement number, *nnn*, that does not exist in routine *routinename*.
Programmer Response: Use a valid statement number.
System Action: VS Pascal does not set a breakpoint.

AMPD502I THERE IS NO ROUTINE NAMED *routinename* IN MODULE *modulename*
Explanation: The routine *routinename* does not exist in module *modulename*.
Programmer Response: Specify correct module and routine names.
System Action: VS Pascal ignores the command.

AMPD503I INVALID QUALIFICATION SPECIFICATION: *token*
Explanation: You incorrectly qualified the command with *token*. This can happen when:
- You fail to use a "/" where required.
- The statement specification you give for a BREAK or RESET command is neither a number nor "END".

Programmer Response: Reissue the command with a valid qualification.
System Action: VS Pascal ignores the command.

AMPD504I MISSING QUALIFICATION SPECIFICATION
Explanation: You omitted part of the qualification for the command.
Programmer Response: Correct the error.
System Action: VS Pascal ignores the command.

AMPD505I MODULE NAME MUST BE SPECIFIED
Explanation: You failed to specify a unit name for a BREAK or RESET command, and you have not specified a default unit name.
Programmer Response: Either:
- Specify a unit name on the command.
- Issue a QUAL command to specify a default unit name.

System Action: VS Pascal ignores the command.

AMPD506I BREAKPOINT IS ALREADY SET
Explanation: The breakpoint you entered is already set.
Programmer Response: If want to modify a breakpoint's options:
1. Issue the RESET command to remove the breakpoint.
2. Issue the BREAK command to set the breakpoint with the new options.

System Action: VS Pascal leaves the breakpoint unchanged.

AMPD507I MAXIMUM NUMBER OF BREAKPOINTS HAVE BEEN SET
Explanation: The maximum number of breakpoints is already set. You cannot set another breakpoint.
Programmer Response: If you want to set a new breakpoint, you must first remove an existing breakpoint.
System Action: VS Pascal does not set the breakpoint.

AMPD508I SPECIFIED BREAKPOINT DOES NOT EXIST
Explanation: You tried to reset a breakpoint that does not exist.
Programmer Response: Ensure that you correctly specified the breakpoint you want to reset. If this is not the problem, the breakpoint you tried to reset has already been reset.
System Action: VS Pascal does not remove any breakpoints.

AMPD509I *Variablename* IS AN AUTOMATIC VARIABLE LOCAL TO A NON-ACTIVE ROUTINE
Explanation: The identifier *variablename* is a local automatic variable in an inactive routine. This error occurs when you try to view an automatic variable in a routine not in the current call chain.
Programmer Response: Ensure that you specified the correct variable name. If you specified the correct variable name, you must wait until the routine becomes active before you can view the the variable's value.
System Action: VS Pascal ignores the command.

AMPD510I FIELD QUALIFIED VARIABLE IS NOT A RECORD
Explanation: You qualified a variable with a field name, but the variable is not a record.
Programmer Response: Either:
- Specify a record variable name to the left of the period.
- Do not qualify the variable.

System Action: VS Pascal ignores the command.

AMPD511I *Identifier* IS NOT A VALID RECORD FIELD
Explanation: The field name you specified, *identifier*, is not valid for the variable.
Programmer Response: Either:
- Use a variable that contains *identifier* as a valid field name.
- Use a field name that is valid for the variable.
- Do not qualify the variable.

System Action: VS Pascal ignores the command.

AMPD512I SUBSCRIPTED VARIABLE IS NOT AN ARRAY
Explanation: You specified a subscript for a variable that is not an array, string, or space.
Programmer Response: Either:
- Do not subscript the variable.
- Specify an indexed variable.

System Action: VS Pascal ignores the command.

AMPD513I ARRAY SUBSCRIPT IS NOT A SCALAR
Explanation: You used a non-scalar value as an index for an array, string, or space variable. You can use only integers, characters, and enumerated scalars as index values.
Programmer Response: Use a scalar value to subscript the indexed variable.
System Action: VS Pascal ignores the command.

AMPD514I INVALID SYMBOL: *symbol*
Explanation: The indicated symbol (*symbol*) is not valid in the context in which it is used.
Programmer Response: Either:
- Remove *symbol* from the command.
- Replace *symbol* with an appropriate symbol.

System Action: VS Pascal ignores the command.

AMPD515I ARRAY SUBSCRIPT IS OUT OF BOUNDS: *subscript*
Explanation: You used an out-of-bounds subscript on an array, string, or space reference. For variable-length strings, the subscript must be less than or equal to the length of the string.
Programmer Response: Use a valid subscript or a different variable.
System Action: VS Pascal ignores the command.

AMPD516I MISSING SYMBOL: *symbol*
Explanation: You must use *symbol* with the command.
Programmer Response: Reissue the command with *symbol* in the proper place.
System Action: VS Pascal ignores the command.

AMPD517I ASSOCIATED VARIABLE IS NOT A POINTER
Explanation: You tried to dereference a variable that is not a pointer.
Programmer Response: Either:
- Do not use the variable as a pointer.
- Use a variable that is a pointer.

System Action: VS Pascal ignores the command.

AMPD518I POINTER VARIABLE DOES NOT CONTAIN VALID ADDRESS
Explanation: You tried to dereference a pointer that does not contain a valid address. This pointer is either undefined or NIL.
Programmer Response: Either:
- Do not try to dereference the pointer.
- Use a valid pointer.

System Action: VS Pascal ignores the command.

AMPD519I *Identifier* **NOT FOUND IN SYMBOL TABLE**
Explanation: You specified *identifier* in a command, but *identifier* is not in the symbol table. The debug symbol table contains only variable identifiers. VS Pascal does not keep information about labels, constants, and types at run time, and you cannot display information about routines, programs, or segments.
Programmer Response: Reissue the command with the correct variable identifier.
System Action: VS Pascal ignores the command.

AMPD520I EQUATE SUBSTITUTION IS IN INFINITE RECURSION
Explanation: You specified a circular series of EQUATE values. When VS Pascal begins substituting the EQUATE values you provided, it enters an endless loop.
Programmer Response: Remove one of the EQUATE values that cause the loop.
System Action: VS Pascal stops expanding the EQUATE values and ignores the input line.

AMPD521I EQUATE EXPANSION CAUSES COMMAND TRUNCATION (EXCEEDS 255 CHARACTERS)
Explanation: The EQUATE values you provided caused a command to expand beyond the 255-character maximum.
Programmer Response: Issue the commands truncated because of the error.
System Action: VS Pascal expands the EQUATE and truncates the result. It then processes the input as is.

AMPD522I YOU ARE NOT IN CMS, COMMAND NOT VALID
Explanation: You issued the CMS command, but the Interactive Debugging Tool is not running in a CMS environment.
Programmer Response: Either:
- Use the Interactive Debugging Tool in a CMS environment.
- Do not use the CMS command.

System Action: VS Pascal does not enter CMS.

AMPD523I DEBUG COMMAND NOT RECOGNIZED: *token*
Explanation: The command you issued, *token*, is either:
- Not a valid debugging command.
- Not equated to a valid debugging command.

Programmer Response: Either:
- Issue a valid debugging command.
- Equate *token* to a valid debugging command.

System Action: VS Pascal ignores the command string, but it processes all input after the command normally. For example, if you issue the command

```
BRAK X/3; GO
```

VS Pascal issues error message AMPD523 because BRAK is not a valid debugging command. VS Pascal then executes the GO command.

AMPD524I INVALID CHARACTER IN HEXADECIMAL STRING: *character*
Explanation: You used *character* in a hexadecimal string, but this is not a valid hexadecimal digit.
Programmer Response: Replace *character* with a valid hexadecimal digit.
System Action: VS Pascal ignores the command.

AMPD525I INVALID HEXADECIMAL STRING
Explanation: You specified an invalid hexadecimal string on a storage-viewing command. Hexadecimal strings consist of:
1. A single quotation mark (')
2. Eight hexadecimal digits
3. Another single quotation mark (')
4. An "X".

For example, '0000917F'X is a valid hexadecimal string. In this case, the last "'" is missing.
Programmer Response: Reissue the command with a valid hexadecimal string.
System Action: VS Pascal ignores the view command.

AMPD526I ROUTINE IS NOT ACTIVE
Explanation: You issued a QUAL command that set the current qualification to an inactive routine.
Programmer Response: Correct any errors in the routine name. If the routine name is correct, make sure that you are trying to view only STATIC, DEF, and REF variables.
System Action: VS Pascal accepts the command, but you cannot view local automatic variables for this routine.

AMPD5271 QUALIFICATION SET TO MODULE
Explanation: You issued a QUAL command that set the current qualification to the global variables of a segment.
Programmer Response: Set the qualification to the program unit's main program to view global variables.
System Action: VS Pascal accepts the command, but no variables are in the qualification.

AMPD5281 THE WORD 'EQUATE' MAY NOT BE REDEFINED
Explanation: You tried to redefine the word "equate" with the EQUATE command. This is not allowed. However, you can use any abbreviation of "equate" in an EQUATE command.
Programmer Response: Choose a different identifier to use in the EQUATE command.
System Action: VS Pascal ignores the EQUATE command.

AMPD5291 MAXIMUM NUMBER OF EQUATES HAVE BEEN SET
Explanation: You tried to create a new equate, but the maximum number of equates is already set.
Programmer Response: Either:
- Dispose of an existing equate.
- Do not enter any new equates.

System Action: VS Pascal leaves the existing list of equates as is.

AMPD5301 THERE ARE NO EQUATES CURRENTLY SET
Explanation: You issued the DISPLAY EQUATES command, but no equates exist.
Programmer Response: Make sure you did not accidentally remove equates that you set previously.
System Action: VS Pascal does not display any equates.

AMPD5311 STATEMENT TABLE MISSING; TRACE REQUIRES GOSTMT OPTION
Explanation: You issued a SET TRACE ON, and program execution resumed. However, VS Pascal did not find a statement table. The statement table is necessary for program tracing.
Programmer Response: Compile the program with the GOSTMT option.
System Action: VS Pascal does not display trace information.

AMPD5321 EQUATE DOES NOT EXIST: *identifier*
Explanation: You tried to remove an equate with the command:

```
EQUATE identifier
```

However, *identifier* is not an equate.
Programmer Response:
- If you want to dispose of an equate, use the correct name of the equate.
- If you want to create a new equate, reissue the command with the correct syntax:

  ```
  EQUATE identifier string
  ```

  where *string* is the text to which the identifier must be equated.

System Action: VS Pascal leaves the existing list of equates as is.

AMPD5331 THERE ARE NO ACTIVE VARIABLES
Explanation: You tried to view a variable, but the currently qualified routine contains no variables.
Programmer Response: Ensure the current qualification is correct.
System Action: VS Pascal does not display a variable.

AMPD5341 ROUTINE IS NOT ACTIVE: *routinename*
Explanation: You used a LISTVARS command for *routinename*, but this routine is not currently active.
Programmer Response: Ensure that the current qualification is correct. If the current qualification is correct, make sure that you used LISTVARS correctly. You can use LISTVARS to display local automatic variables. To display STATIC, DEF, or REF variables, you must issue a view command instead.
System Action: VS Pascal does not display a variable.

AMPD5351 RESET DOES NOT ALLOW ASSOCIATED COMMANDS
Explanation: You tried to use an associated command with RESET. You can use RESET only to remove a breakpoint. You cannot use RESET to change a command associated with a breakpoint.
Programmer Response:
- If you want only to remove the breakpoint, issue the RESET command without the associated command.
- If you want to change the command associated with a breakpoint:
  1. Issue the RESET command without the associated command.
  2. Use the BREAK command to specify a new associated command.

System Action: VS Pascal ignores the RESET command. The breakpoint remains set.

AMPD5361 BREAKPOINT COMMAND IGNORED; COMMAND WOULD EXCEED 255 CHARS
Explanation: A command associated with a breakpoint would have caused the current input line to exceed the maximum of 255 characters. This error should occur only when an input line is not completely exhausted following a GO or WALK command.
Programmer Response: If no commands are pending, you can execute the commands associated with the breakpoint.
System Action: VS Pascal ignores the associated command and processes the remainder of the input line.

AMPD5371 VIEW ONLY WORKS WITH IDENTIFIERS OR HEX NUMBERS
Explanation: The variable you specified on a view command is neither an identifier nor a hexadecimal storage address.
Programmer Response: Specify an identifier or hexadecimal storage address after the comma (,) in the view command.
System Action: VS Pascal ignores the view command.

AMPD5381 FILE ID EXPECTED
Explanation: You failed to specify a file identifier.
Programmer Response: Reissue the command with a file identifier in the proper place.
System Action: VS Pascal ignores the command.

AMPD5391 MODULE NAME EXPECTED
Explanation: You failed to specify a module name.
Programmer Response: Reissue the command with a module name in the proper place.
System Action: VS Pascal ignores the command.

AMPD540I IDENTIFIER EXPECTED
Explanation: You failed to specify an identifier.
Programmer Response: Reissue the command with an identifier in the proper place.
System Action: VS Pascal ignores the command.

AMPD541I INVALID ARGUMENT ON EVERY OPTION OF BREAK
Explanation: You specified an invalid value for the EVERY option on a BREAK command. The EVERY option specifies an increment value. This increment value must be a positive integer.
Programmer Response: Issue the BREAK command with a valid increment value for EVERY.
System Action: VS Pascal ignores the BREAK command and does not set a breakpoint.

AMPD542I INVALID ARGUMENT ON FROM OPTION OF BREAK
Explanation: You specified an invalid value for the FROM option on a BREAK command. The FROM option specifies an initial value. This initial value must be a positive integer or zero.
Programmer Response: Issue the BREAK command with a valid initial value for FROM.
System Action: VS Pascal ignores the BREAK command and does not set a breakpoint.

AMPD543I INVALID ARGUMENT ON TO OPTION OF BREAK
Explanation: You specified an invalid value for the TO option on a BREAK command. The TO option specifies a final value. This final value must be a positive integer greater than or equal to the initial value specified for the FROM option.
Programmer Response: Issue the BREAK command with a valid final value.
System Action: VS Pascal ignores the BREAK command and does not set a breakpoint.

AMPD544I OPTION NOT APPLICABLE ON RESET
Explanation: You specified an EVERY, FROM, or TO option on a RESET command. These options work only with the BREAK command. You can use RESET only to remove a breakpoint. You cannot use RESET to change the value of BREAK options.
Programmer Response:

- If you want to remove the breakpoint, issue the RESET command without the option.
- If you want to change the BREAK option value:
  1. Issue the RESET command without the option.
  2. Issue the BREAK command to specify a new option value.

System Action: VS Pascal ignores the RESET command and leaves the breakpoint set.

AMPD545I LENGTH OF STRING (GSTRING) IS NEGATIVE
Explanation: The length of the STRING (or GSTRING) variable is negative. This can occur when an uninitialized variable is used or when part of a variant record overlays the variable.
Programmer Response: Make sure the length of the variable is greater than or equal to zero.
System Action: VS Pascal does not display the variable.

EXEC Messages

The following messages are issued by VS Pascal EXECs (VSPASCAL, PASCMOD, and PASCRUN).

AMPE100E UNABLE TO FIND "*filename filetype filemode*"
Explanation: VS Pascal did not find the program named *filename filetype filemode*.
Programmer Response: Check the spelling of the program name and make sure it is on an accessed disk.
System Action: The compiler does not start.

AMPE101E BAD PARAMETERS SPECIFIED: "*parms*"
Explanation: You specified *parms* after what should be the filemode.
Programmer Response: Either:
- Place a left parenthesis, "(", before any compile-time options you pass to the compiler.
- Remove the bad parameters.

System Action: The compiler does not start.

AMPE102E NO OPTIONS WERE SPECIFIED AFTER THE LEFT PARENTHESIS "(".
Explanation: You used a left parenthesis, "(", on the command line. This indicates that compile-time options follow, but you did not specify any options.
Programmer Response: Either:
- Remove the left parenthesis.
- Specify the desired compile-time options.

System Action: The compiler does not start.

AMPE103E DATA FOUND PAST COMMAND END: "*data*"
Explanation: You specified *data* after the right parenthesis, ")" on a command line. This is invalid, because the right parenthesis ends the command.
Programmer Response: Either:
- Check for mismatched parentheses.
- Move the last right parenthesis after *data*.

System Action: The compiler does not start.

AMPE104E UNABLE TO FIND THE "*libname*" MACLIB
Explanation: VS Pascal did not find the MACLIB named *libname*.
Programmer Response: Check the spelling of the MACLIB name and make sure it is on an accessed disk.
System Action: The compiler does not start.

AMPE105E MORE THAN *n* MACLIBS SPECIFIED
Explanation: You can specify up to *n* MACLIBs when invoking the VSPASCAL EXEC.
Programmer Response: Use fewer MACLIBs. You can try combining several MACLIBs into one.
System Action: The compiler does not start.

AMPE106I CONFLICTING LISTING OPTION: *option*
Explanation: The option *option* conflicts with a previously specified option.
Programmer Response: Check the options specified to ensure they do not conflict with each other.
System Action: The option is ignored and the compiler continues.

AMPE107E SUBOPTION EXPECTED FOR *option*
Explanation: *Option* requires a qualifying suboption, but you did not specify one. The suboption might be missing a left parenthesis.
Programmer Response: Either:
- Do not specify the option at all.
- Specify a valid suboption.

System Action: The compiler does not start.

AMPE108E TOO MANY SUBOPTIONS SPECIFIED AFTER THE *option* OPTION
Explanation: You specified too many suboptions for *option*. You might have omitted a right parenthesis from the suboption list.
Programmer Response: Either:
- Do not specify the option at all.
- Specify a valid suboption.

System Action: The compiler does not start.

AMPE109E *Token* IS NOT A VALID SUBOPTION FOR *option*
Explanation: *Token* is not a valid suboption for compile-time option *option*.
Programmer Response: Either:
- Do not specify the option at all.
- Use a valid suboption.

System Action: The compiler does not start.

AMPE110I MESSAGE TEXT NOT FOUND FOR LANGUAGE: *language*
Explanation: Message text does not exist for *language*, as specified on the LANGUAGE option.
Programmer Response: Change the LANGUAGE option to a language that is supported by VS Pascal and implemented on your system.
System Action: The compiler continues; all output will appear in the default language selected during installation.

AMPE111S A-DISK EITHER NOT ACCESSED OR LINKED READ-ONLY
Explanation: The compiler must write several files to the A-disk. Either:
- The compiler did not find the A-disk.
- The A-disk the compiler found is linked as read-only.

Programmer Response: Either:
- Access the A-disk.
- Link the A-disk in read-write mode.

System Action: The compiler does not start.

AMPE112S **FILE** *filename filetype* **FOUND ON READ-ONLY EXTENSION OF DISK** *filemode*
Explanation: VS Pascal must update the file called *filename filetype*, but the file resides on a read-only extension of disk *filemode*.
Programmer Response: Release the read-only extension by either:

- Releasing the disk.
- Reaccessing the disk so that it is not a read-only extension of disk *filemode*.

System Action: The compiler does not start.

AMPE200E **INVALID OPTIONS:** *options*
Explanation: You invoked the PASCMOD EXEC with invalid options.
Programmer Response: Either:

- Use a valid option.
- Fix the syntax of the options.

System Action: The PASCMOD EXEC ends. The load module is not generated.

AMPE201E **LOAD COMMAND FAILED. RETURN CODE =** *returncode*
Explanation: The LOAD command failed. The return code is *returncode*.
Programmer Response: Refer to the *CMS Command and Macro Reference* for an explanation of the error.
System Action: The EXEC ends. The load module is not generated.

AMPE202E **GLOBAL TXTLIB COMMAND FAILED. RETURN CODE =** *returncode*
Explanation: The GLOBAL TXTLIB command failed. The return code is *returncode*.
Programmer Response: Refer to *CMS Command and Macro Reference* for an explanation of the error.
System Action: The EXEC ends. The load module is not generated.

AMPE203E **GENMOD COMMAND FAILED. RETURN CODE =** *returncode*
Explanation: The GENMOD command failed. The return code is *returncode*.
Programmer Response: Refer to *CMS Command and Macro Reference* for an explanation of the error.
System Action: The EXEC ends. The load module is not generated.

AMPE204E **NO OBJECT FILES OR LIBRARIES WITH THESE NAMES:**
Explanation: VS Pascal cannot find the object files or libraries you specified.
Programmer Response: Check the spelling of the object file or library names and make sure they are on an accessed disk.
System Action: The EXEC ends. The load module is not generated.

AMPE205E **XA OPTION IS NOT SUPPORTED IN VM/SP ENVIRONMENT**
Explanation: You cannot use the XA option when link-editing a program in a VM/SP environment.
Programmer Response: Either:

- Remove the XA option.
- Execute the EXEC under VM/XA.

System Action: The EXEC ends. The load module is not generated.

AMPE206E **GLOBAL LOADLIB COMMAND FAILED. RETURN CODE =** *returncode*
Explanation: The GLOBAL LOADLIB command failed. The return code is *returncode*.
Programmer Response: Refer to the *CMS Command and Macro Reference* for an explanation of the error.
System Action: The EXEC ends. The GLOBAL command is not in effect.

CLIST Messages

The following messages are issued by VS Pascal CLISTs (VSPASCAL and PASCMOD).

AMPC100W **BOTH** *option1* **AND** *option2* **SPECIFIED,** *option* **ASSUMED.**
Explanation: *Option1* and *option2* are conflicting options.
Programmer Response: Correct the options specified to ensure they do not conflict with each other.
System Action: *Option* is used and the compiler continues.

AMPC101W *Token* **IS NOT A VALID SUBOPTION FOR** *option*
Explanation: *Token* is not a valid suboption for compile-time option *option*.
Programmer Response: Either:

- Do not specify the option at all.
- Use a valid suboption.

System Action: The option is ignored and the compiler continues.

AMPC102I **MESSAGE TEXT NOT FOUND FOR LANGUAGE:** *language*
Explanation: Message text does not exist for *language*, as specified on the LANGUAGE option.
Programmer Response: Change the LANGUAGE option to a language that is supported by VS Pascal and implemented on your system.
System Action: The compiler continues; all output will appear in the default language selected during installation.

Appendix A. Summary of Changes

VS Pascal Release 2 provides additions and enhancements to VS Pascal Release 1 in the following areas:

- **System Flexibility**

  VS Pascal now:

  - *Runs under VM/Extended Architecture (VM/XA).*
  - *Runs under MVS/Enterprise Systems Architecture (MVS/ESA).*
  - *Allows communication with other IBM licensed programs, such as IMS.*

- **Communication with Other Programming Languages**

  VS Pascal now:

  - *Supports communication with OS PL/I Version 2.*
  - *Provides better error detection in Assembler routines called by VS Pascal.* Program checks in Assembler routines coded with the PROLOG macro will be handled using the ONERROR procedure used by VS Pascal instead of causing the severe error message AMPX902S. In order for program checks to be handled by ONERROR, Assembler routines using the PROLOG macro need to be reassembled.

- **Transient Run-Time**

  Users now have the option to:

  - *Compile load modules for standard link-editing, or compile modules that will access run-time routines dynamically at execution.* The transient run-time library option helps free resources in large-scale, modular systems that must serve multiple users. Transient run-time reduces the size of load modules and makes it unnecessary for each program to include a copy of the run-time library.

- **Compiler Features**

  Users now have the option to:

  - *Compile only selected portions of a source program.* This "conditional compilation" feature simplifies debugging and supports multiple operating environments.

  - *Place headers in generated code.* Headers include the name of the compiled routine, the compiler name, and the date and time of compilation. Users can also insert a customized header after the compiler header.

- **Compile-Time Limits**

  Users can now write and debug larger and more complex programs. Each compilable program can have up to:

  - *999 %INCLUDE directives* (previous limit: 255).
  - *8192 TYPE declarations* (previous limit: 255).
  - *32678 characters in identifier names in a routine* (previous limit: 8192).
  - *1024 fields per record* (previous limit: 255).

- **Debugging**

  Users can now:

  - *Invoke the interactive debugging tool without having to issue further debugging instructions.* With this option, the debugger prompts users for further instructions only when it detects an error. With this "hot debugger," less experienced users can test programs in a production environment.
  - *Specify how many instances of a breakpoint can occur before program execution halts.* Previously, execution halted at every occurrence of a breakpoint. Programmers now have the flexibility of bypassing a specified number of breakpoint occurrences at a repeated statement.
  - *Display the statistics kept by the COUNT run-time option at any time during a debugging session.*

- **Error Handling**

  VS Pascal Release 2 now:

  - *Checks compiler directives for syntax, semantic, and limit errors.*
  - *Finishes printing its cross-reference and statistics listing when it encounters a severe error.* Previously, severe errors caused the listing to abort immediately.
  - *Flags invalid compile-time option syntax and suboptions that were previously ignored.*

- **Storage Considerations**

  VS Pascal Release 2 allows users to tune programs by providing:

  - *Enhanced heap management options.* Users can now control the initial size of the default heap, in addition to the amount it is extended on overflow.
  - *Multiple heap support.* To help alleviate storage fragmentation problems, a component of a large, multicomponent program can now create and manage its own heap independent of heaps associated with other components.

- **Double-Byte Character Set (DBCS) Data**

  Among many new DBCS features, Release 2 supports:

  - *A predefined scalar data type, GCHAR, which represents one DBCS character.*
  - *A predefined structured data type, GSTRING, which represents a DBCS string.*
  - *Hexadecimal graphic data.*

  Existing string manipulation routines were revised for DBCS support, and special DBCS routines for handling mixed strings were added:

  - *Many existing string routines now work with GSTRING data in a character-oriented manner (two bytes at a time).*
  - *New string routines were added which work with SBCS and DBCS STRING data in a character-oriented (not byte-oriented) manner.*

- **Proposed ANSI/IEEE Extended Pascal**

  VS Pascal Release 2 adds support for:

  - *The EPSREAL predefined constant.*
  - *The MAXCHAR predefined constant.*
  - *A plus sign (+) for string concatenation (||).*
  - *A set symmetric difference operator (><).*

- **National Language Support**

  Release 2 also:

  - *Adopts the Syntactic Graphic Character Set (GCSGID 640) as its standard character set.* VS Pascal programs do not require characters outside this set, establishing a standard that makes programs easily transportable between different sites.
  - *Allows customization of character translation and uppercase tables at installation.* This eases compiler recognition of tokens and characters due to national programming standards and allows the creation of uppercase rules.
  - *Provides three languages from which sites choose a default language during installation.* At both run time and compile time, users can override the default language with another language. Currently, VS Pascal provides mixed-case English, uppercase English, and Japanese.

Appendix B. How to Read Syntax Diagrams

Read syntax diagrams from left to right, top to bottom.

```
►►———   indicates the beginning of the diagram.
———►    indicates that the syntax is continued on the next line.
►———    indicates that the syntax is continued from the previous line.
———►◄   indicates the end of the diagram.
```

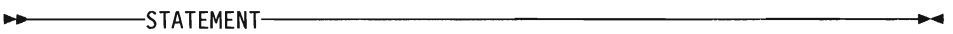
Keywords appear in all capital letters. For example: VAR, BEGIN, END. When writing code, enter keywords exactly as shown, either in all caps or in lowercase.

Variables appear in lowercase in a special typeface. For example: *label-dcl*.

Special Symbols such as " > ", " = ", and so forth must be entered as part of the code.

No Parameters

A keyword that requires no parameter is diagrammed this way:



Remember that you must code keywords (those in capital letters) exactly as shown, or in lowercase. In this example, you can code

```
STATEMENT
```

or

```
statement
```

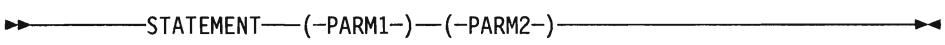
Required Parameters

All required keywords and variables appear on the diagram's main path. In this example,

you must code both parameters. Always separate parameters with one or more blanks.

```
STATEMENT PARM1 PARM2
```

Parentheses around parameters, like all special symbols, must be coded exactly as shown. In this example,



you must code:

```
STATEMENT (PARM1) (PARM2)
```

When there is a vertical list of parameters, one of which is on the main path, you *must* choose only one of them. In this example,

you must code:

```
STATEMENT PARM1
```

or

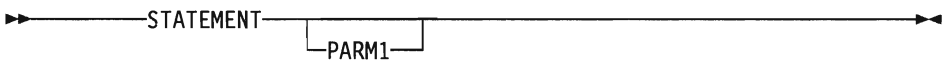
```
STATEMENT PARM2
```

or

```
STATEMENT PARM3
```

Optional Parameters

A single optional parameter appears below the main path. In this example,



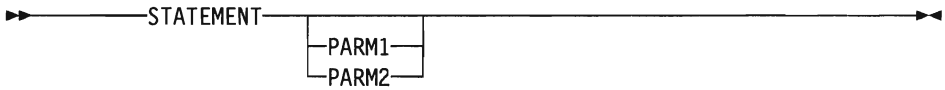
you must code either:

```
STATEMENT
```

or

```
STATEMENT PARM1
```

When you can choose only one optional parameter from a list of two or more, the choices appear in a vertical list below the main path. In this example,



you must code:

```
STATEMENT
```

or

```
STATEMENT PARM1
```

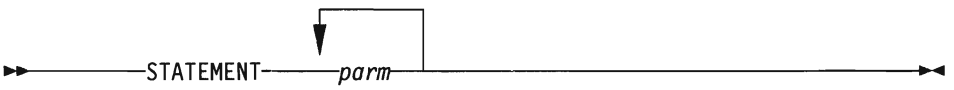
or

```
STATEMENT PARM2
```

Multiple Parameters

The repeat symbol

indicates that you can specify more than one parameter or a single parameter more than once. In this example,



parm, shown in small letters, represents a variable parameter. If the values you can substitute for *parm* include PARM1 and PARM2, then you can code:

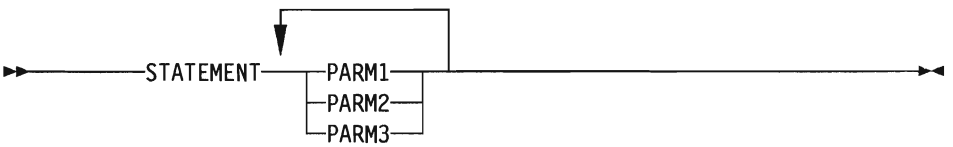
```
STATEMENT parm1
```

or

```
STATEMENT parm1 parm2
```

and so forth.

This diagram



indicates that you can code:

```
STATEMENT PARM1
```

or

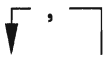
```
STATEMENT PARM1 PARM3
```

or

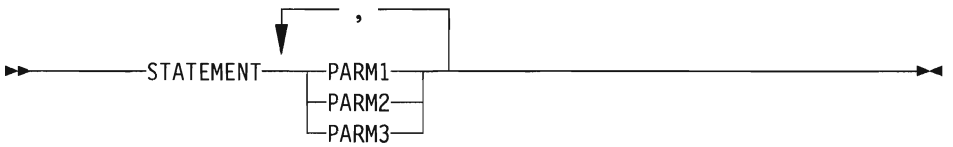
```
STATEMENT PARM1 PARM2 PARM3
```

and so forth.

When the repeat symbol contains a comma,



you must separate multiple parameters with commas. In such cases, parameters need not be separated by blanks. In this example,



you can code:

```
STATEMENT PARM1
```

or

```
STATEMENT PARM1, PARM3
```

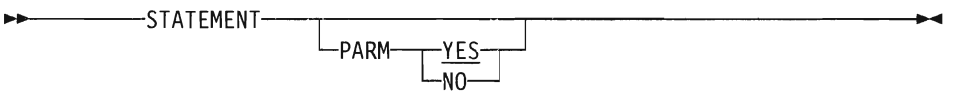
or

```
STATEMENT PARM1,PARM2,PARM3
```

and so forth.

Default Parameters

Default parameters are underscored. Omitting a parameter with a default value produces the same result as actually coding the default. In this example,



coding

```
STATEMENT PARM
```

is equivalent to coding

```
STATEMENT PARM YES
```

Appendix C. Run-Time Error Default Actions

The following table lists the defaults for run-time error messages passed to ONERROR in FACTION.

Message			Defaults		
AMPX011	XHALT	XPMSG	XTRACE		XDECERR
AMPX012	XHALT	XPMSG	XTRACE		XDECERR
AMPX013	XHALT	XPMSG	XTRACE		XDECERR
AMPX014	XHALT	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX015	XHALT	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX016	XHALT	XPMSG	XTRACE		XDECERR
AMPX017	XHALT	XPMSG	XTRACE		XDECERR
AMPX018		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX019		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX020		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX021		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX022		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX023		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX024		XPMSG	XTRACE		XDECERR
AMPX025		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX026	XHALT	XPMSG	XTRACE		XDECERR
AMPX027	XHALT	XPMSG	XTRACE		XDECERR
AMPX028	XHALT	XPMSG	XTRACE		XDECERR
AMPX029	XHALT	XPMSG	XTRACE		XDECERR
AMPX030		XPMSG	XTRACE	XDEBUG	
AMPX031		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX032		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX033		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX034		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX035		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX036		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX037		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX038		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX039		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX040		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX041	XHALT	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX042		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX043		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX044		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX045		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX046		XPMSG	XTRACE	XDEBUG	XDECERR
AMPX048		XPMSG	XTRACE	XDEBUG	XDECERR

Figure 109 (Part 1 of 3). Defaults for Messages Passed to ONERROR in FACTION

Message	Defaults			
AMPX049	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX050	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX051	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX052	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX053	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX054	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX055	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX056	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX058	XPMSG	XTRACE		
AMPX059	XPMSG	XTRACE		XDECERR
AMPX060	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX061	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX062	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX063	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX064	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX065	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX066	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX067	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX068	XPMSG	XTRACE		XDECERR
AMPX069	XPMSG	XTRACE		XDECERR
AMPX070	XPMSG	XTRACE		XDECERR
AMPX071	XPMSG	XTRACE		XDECERR
AMPX072	XPMSG	XTRACE		XDECERR
AMPX073	XPMSG	XTRACE		XDECERR
AMPX074	XPMSG	XTRACE		XDECERR
AMPX075	XPMSG	XTRACE		XDECERR
AMPX076	XPMSG	XTRACE		XDECERR
AMPX077	XPMSG	XTRACE		XDECERR
AMPX078	XPMSG	XTRACE		XDECERR
AMPX079	XPMSG	XTRACE		XDECERR
AMPX080	XPMSG	XTRACE		XDECERR
AMPX081	XPMSG	XTRACE		XDECERR
AMPX082	XPMSG	XTRACE		XDECERR
AMPX083	XPMSG	XTRACE		XDECERR
AMPX084	XPMSG	XTRACE		XDECERR
AMPX085	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX090	XPMSG	XTRACE		XDECERR
AMPX091	XPMSG	XTRACE		XDECERR
AMPX092	XPMSG	XTRACE		XDECERR
AMPX093	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX094	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX095	XPMSG	XTRACE	XDEBUG	XDECERR

Figure 109 (Part 2 of 3). Defaults for Messages Passed to ONERROR in FACTION

Message	Defaults			
AMPX096	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX097	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX098	XPMSG	XTRACE	XDEBUG	XDECERR
AMPX099	XPMSG	XTRACE		XDECERR
AMPX100	XPMSG	XTRACE		XDECERR
AMPX101	XPMSG	XTRACE		XDECERR
AMPX102	XPMSG	XTRACE		XDECERR
AMPX600	XPMSG	XTRACE		XDECERR
AMPX601	XPMSG	XTRACE		XDECERR
AMPX602	XPMSG	XTRACE		XDECERR
AMPX603	XPMSG	XTRACE		XDECERR
AMPX604	XPMSG	XTRACE		XDECERR
AMPX605	XPMSG	XTRACE		XDECERR
AMPX700	XPMSG	XTRACE		XDECERR
AMPX701	XPMSG	XTRACE		XDECERR
AMPX702	XPMSG	XTRACE		XDECERR
AMPX703	XPMSG	XTRACE		XDECERR

Figure 109 (Part 3 of 3). Defaults for Messages Passed to ONERROR in FACTION

Appendix D. VS Pascal and the 1983 ANSI/IEEE Pascal Standard

The VS Pascal (MVS version) processor complies with the requirements of ANSI/IEEE770X3.97-1983.

The VS Pascal (MVS version) processor complies with the requirements of level 0 of ISO 7185.

The VS Pascal (VM version) processor complies with the requirements of ANSI/IEEE770X3.97-1983 with the following exceptions:

- Due to an operating system limitation, empty lines in text files are not supported (such as a line which has EOLN true immediately upon reading it in).
- Due to an operating system limitation, empty files are not supported (such as a file which has EOF true immediately after applying RESET to it).

The VS Pascal (VM version) processor complies with the requirements of level 0 of ISO 7185 with the same exceptions as noted above.

The remainder of this section discusses the implementation-defined features of VS Pascal and how VS Pascal handles errors, extensions and implementation-dependent features.

Implementation-Defined Features of VS Pascal

The definitions of implementation-defined features of VS Pascal are listed below. For information on the EBCDIC character set and System/370 floating-point type, see the appropriate manual in "Principles of Operation" on page 296.

- The characters allowed in string types are the printable EBCDIC characters plus the space character. Depending on the system and terminal being used, different characters may be visible.
- The character set used by the data type CHAR is the EBCDIC character set. Depending on the system and terminal being used, different characters may be visible.
- The ordinal values used for each character in the CHAR data type are those given by the EBCDIC character set. Depending on the terminal and system being used, the same ordinal value may correspond to different characters.
- The subset of real numbers used by the VS Pascal REAL data type is the same as the System/370 long floating-point type.
- The accuracy of real arithmetic is that given by the System/370 hardware long floating-point operations.
- The value of MAXINT is 2147483647.
- The actions taken by the file handling procedures and the time of their performance are specified below.

- REWRITE—REWRITE(*f*) causes the external file bound to *f* to be erased, and the file to be opened for output. These operations are performed immediately upon executing the REWRITE.
- PUT—PUT(*f*) causes the value of the file buffer to be placed into a special buffer. This buffer is written to the external file upon any of the following occurrences:
  - The special buffer becomes full (this buffer's capacity is equal to the physical file's block size).
  - A RESET is performed on file *f*.
  - The routine or program containing the file *f* ends.
  - In LANGLVL(EXTENDED), the CLOSE, UPDATE, TERMIN, TERMOUT, PDSIN, and PDSOUT procedures are applied to file *f*.
- RESET—RESET(*f*) causes the external file bound to *f* to open for input. This occurs immediately upon executing the RESET. The first element in the external file bound to *f* is immediately assigned to the file buffer, unless (in LANGLVL(EXTENDED) only) the file is an interactive file. In this case, the file pointer has the value NIL until the first GET or READ is done on the file. At this time, the file buffer will contain the first element in the file.
- GET—GET(*f*) causes the file pointer to point to the next unread element of *f* (if any), and the file buffer to be updated with that element. This action occurs immediately upon execution of the GET.

- The default field width for integer output is 12.
- The default field width for Boolean output is 10.
- Boolean variables are output in all upper case (such as TRUE and FALSE).
- The default field width for real output is 20.
- The exponent character for real output is E.
- Two exponent digits are used for real output.
- The effect of the PAGE procedure on a TEXT file depends on the type of external file the TEXT file is bound to. If the TEXT file is bound to an ASA (RECFM A) file, a "1" is written in column one (the carriage control column) of the next line of the file. If the TEXT file is bound to a machine (RECFM M) file, a X'8B' is written in column one of the next line of the file. A new line is started after this. If neither of the above conditions hold true, the PAGE procedure has no effect on the file other than performing the WRITELN specified by the ANSI/IEEE Pascal standard.
- Files listed as program parameters are bound to external files in the same way as any other file.

  Note: The first 8 characters of the program parameter's name will form the ddname of the file, regardless of the setting of the DDNAME compile-timer option.
- Applying RESET and REWRITE to TEXT files INPUT and OUTPUT will yield the same results as applying RESET and REWRITE to any other file.

Error Handling in VS Pascal

This section describes how VS Pascal handles the errors listed in the 1983 ANSI/IEEE Pascal standard. The following list parallels the error handling list in Appendix D of that standard.

It is assumed that all compile-time and run-time checking has been turned on. If the checking is off, some errors that would normally be caught will not be caught. Also, if an error occurs in a program, further errors may or may not be caught.

Unless an error is stated to be caught (with no qualifications), for the purposes of the standard, it may be considered to be undetected. Thus, if an error is stated to be "generally caught", there are some circumstances where it will be left undetected.

1. Array subscripting errors are caught.
2. Accessing a component of an inactive variant is not caught (for RECORD data type).
3. NIL pointer dereferencing errors are caught.
4. Undefined pointer dereferencing errors are not caught except in special circumstances.
5. Removing the identifying value of an identified (pointer-qualified) variable while a reference to that variable exists is not caught.
6. Altering the value of a file variable while a reference to its buffer variable exists is not caught.
7. Assignment compatibility errors between actual and formal ordinal value parameters are caught.
8. Assignment compatibility errors between actual and formal set value parameters are caught.
9. Using the PUT, WRITE, WRITELN, and PAGE procedures on files not open for output is generally caught.
10. Using the PUT, WRITE, WRITELN, and PAGE procedures on undefined files is generally caught.
11. Using the PUT, WRITE, WRITELN, and PAGE procedures on files with end-of-file not true before the call is not possible with LANGLVL(ANSI83). For LANGLVL(EXTENDED), the UPDATE procedure and random access I/O allow the end-of-file to be false.
12. Using the PUT procedure with an undefined buffer variable is not caught.
13. Using the RESET procedure on an undefined file is not caught.
14. Using the GET, READ, and READLN procedures on files that are not open for input is generally caught.
15. Using the GET, READ, and READLN procedures on undefined files are generally caught.
16. Attempting to use the GET, READ, or READLN procedures with end-of-file true is caught.
17. For READ, assignment compatibility errors are caught.
18. For WRITE, assignment compatibility errors are caught.

19. Activating a variant of a variant part of a variable created with the NEW procedure of the form (*p,t1,...,tn*) where the variant is in the same variant part as, but different from, one of the specified variants is not caught.
20. Using the DISPOSE(*p*) procedure on a variable created with the NEW procedure of the form (*p,t1,...,tn*) is not caught.
21. Using the DISPOSE procedure of the form (*p,t1,...,ty*) on a variable created with the NEW procedure of the form (*p,t1,...,tx*) where *x* is not equal to *y* is not caught.
22. Using the DISPOSE procedure of the form (*p,t1,...,tn*) on a variable with active variants different from (*t1,...,tn*) is not caught.
23. Using the DISPOSE procedure on a NIL pointer is caught.
24. Using the DISPOSE procedure on an undefined pointer is not generally caught.
25. Accessing variables created by the NEW procedure of the form (*p,t1,...,tn*) using the identified variable in a factor, an assignment statement, or as an actual parameter is not caught.
26. Assignment compatibility errors on the ordinal variable in the PACK procedure are caught.
27. Accessing undefined elements of the unpacked array in the PACK procedure is not caught.
28. Exceeding the index of the unpacked array in the PACK procedure is caught.
29. Assignment compatibility errors on the ordinal variable in the UNPACK procedure are caught.
30. Having undefined elements in the packed array in the UNPACK procedure is not caught.
31. Exceeding the index of the unpacked array in the UNPACK procedure is caught.
32. Applying the SQR function to a value whose square would not exist is generally caught.
33. Applying the LN function to a value less than or equal to zero is caught.
34. Applying the SQRT function to a negative value is caught.
35. Applying the TRUNC function to a value that would produce an integer with too large a magnitude is generally caught.
36. Applying the ROUND function to a value that would produce an integer with too large a magnitude is generally caught.
37. Applying the CHR function to a value that does not have a character value is caught.
38. Applying the SUCC function to a value that does not have a successor is not caught.
39. Applying the PRED function to a value that does not have a predecessor is not caught.
40. Using the EOF function on an undefined file is not caught.
41. Using the EOLN function on an undefined file is not caught.
42. Using the EOLN function on a file with the end-of-file condition true is not caught.

43. Using undefined variables in an expression is not caught.
44. Real division by zero is caught.
45. Integer division by zero is caught.
46. Using the MOD operator with a second operand less than or equal to zero is caught.
47. Not performing integer operations or functions according to the mathematical rules for integer arithmetic is generally caught.
48. Undefined function result errors are caught.
49. Assignment compatibility errors with ordinal types are caught.
50. Assignment compatibility errors with set types are caught.
51. CASE statement index errors are caught.
52. Assignment compatibility errors between FOR loop indices and initial values are caught.
53. Assignment compatibility errors between FOR loop indices and final values are caught.
54. Attempting to read an integer from a TEXT file where the sequence of characters, after skipping preceding spaces and end-of-line conditions, does not form a signed integer is caught.
55. Type compatibility for integers read from TEXT files is checked and caught.
56. Attempting to read a number from a TEXT file where the sequence of characters, after skipping preceding spaces and end-of-line conditions, does not form a signed number is caught.
57. Using the READ or READLN procedure on a file with the buffer variable undefined is caught. This error can occur when trying to read past the end of file.
58. Using the form "*e : length1 : length2*" or "*e : length1*" in the WRITE and WRITELN procedures for TEXT files where *length1* or *length2* are less than one, is caught (in LANGLVL(ANSI83) only).

Extension Handling

When using the LANGLVL(ANSI83) and DDNAME(UNIQUE) compiler options, no extensions to the ANSI/IEEE standard are allowed. When using the LANGLVL(EXTENDED) compile-time option, all extensions listed in *VS Pascal Language Reference* may be used. When using the DDNAME(COMPAT) compile-time option, all file variables (not in a structure or on the heap) map to ddnames based on the first 8 characters of their names.

Implementation-Dependent Features Not Flagged

Because most implementation-dependent features are used frequently in VS Pascal, these features will not be flagged. The following implementation-dependent features are not flagged by VS Pascal:

- The order of evaluation of array indices.
- The order of evaluation of members in a set constructor.

- The order of evaluation of component expressions in a member-designator in a set constructor.
- The order of evaluation of the operands of a dyadic operator (such as A + B).
- The order of evaluation of the actual parameters in a procedure or function call.
- The order of evaluation of the left-hand side and the right-hand side of an assignment statement.
- The effect of reading from a file that had the PAGE procedure applied to it.
- The binding of program parameters not declared as files.

Appendix E. Implementation Specifics

This section discusses various VS Pascal implementation dependencies. Note that the compiler limits are approximations. Your program may deviate from the maximum limits depending on the complexity of your program.

Routines That May Not Be Passed As Parameters

VS Pascal does not allow the following standard routines to be passed as parameters to another routine:

ABS	GTOSTR	MIN	READSTR	SUBSTR
ADDR	HBOUND	NEW	RELEASE	SUCC
CHR	HIGHEST	ODD	RESET	TERMIN
CLOSE	INDEX	ORD	REWRITE	TERMOUT
COMPRESS	LBOUND	PACK	RINDEX	TRIM
DELETE	LENGTH	PAGE	ROUND	TRUNC
DISPOSE	LOWEST	PDSIN	RPAD	UNPACK
EOF	LPAD	PDSOUT	SEEK	UPDATE
EOLN	LTRIM	PRED	SIZEOF	WRITE
FLOAT	MARK	PUT	SQR	WRITELN
GET	MAX	READ	STOGSTR	WRITESTR
GSTR	MAXLENGTH	READLN	STR	

GENERIC procedures and FORTRAN functions or subroutines may not be passed as parameters to a VS Pascal routine.

Data Types

INTEGER Data Type

The largest integer that may be represented is 2147483647. This is the highest signed value that may be represented in a 32-bit word. This is the value of the predefined constant MAXINT.

Floating-Point Arithmetic

Figure 110 shows some commonly required characteristics of System/370 floating-point arithmetic.

Characteristic	Decimal Approximation	Exact Representation (see note 1)
MAXREAL (see note 2)	7.23700557733226E+75	'7FFFFFFFFFFFFFFF'XR
MINREAL (see note 3)	5.39760534693403E-79	'0010000000000000'XR
EPSREAL (see note 4)	1.38777878078145E-17	'3310000000000000'XR

Figure 110. Characteristics of System/370 Floating-Point Arithmetic

Notes to Figure 110 :

1. The syntax '...'XR is the way hexadecimal floating-point numbers are represented in VS Pascal. See the section titled "Constants" in *VS Pascal Language Reference*.
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. EPSREAL is the smallest positive floating-point number such that the following condition holds:

   ```
   1.0+EPSREAL > 1.0
   ```

   Its value is in the predefined constant EPSREAL. This value is often needed in numerical computations involving converging series.

SET Data Type

Given a SET data 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 true:

```
ORD(a) >= 0
ORD(b) <= 255
```

Compiler Limits

Routine Nesting

Routines may be nested up to eight levels deep including the main program.

Identifiers

Because identifiers cannot span multiple lines, and because VS Pascal restricts the length of an input line to 100 characters or less, VS Pascal will not allow identifiers to be longer than 100 characters.

External procedures and DEF and REF variables may not start with the following letters (because they are used internally by VS Pascal):

- AMPD
- AMPX
- AMPY
- AMPZ

Also, the following external routine names may not be used:

- CMS
- ITOHS
- LPAD
- PICTURE
- RPAD

Size Limitations

Source code limits

- The maximum number of identifiers allowed in a unit is 65536.
- The maximum number of %INCLUDE directives allowed in a unit is 999.
- Units are limited to 65535 lines of code.

Data size limits

- Variables are limited to 16 megabytes.
- The size of all VAR variables in a routine must be less than 16 megabytes.
- The size of all STATIC variables in a unit must be less than 16 megabytes.
- Sets are limited to 256 members with ordinal values 0 through 255.

Debugging limits

- The Interactive Debugging Tool types limit is 8192.
- The maximum number of characters in variable names in a routine is 32767—*n**2, where *n* is the number of variables in the routine.
- Records are limited to 1024 fields.

Code size limits

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

Operating system limits

- The combined number of routines and DEF/REF variables allowed is 32767.

Appendix F. Double-Byte Character Set (DBCS) Support

Many written languages contain a large number of characters. For example, Japanese newspapers use as many as 4000 different characters; a Chinese scholar may recognize as many as 40000 characters; the Korean and Thai languages also have thousands of characters for everyday usage.

English is often used within the data processing environment of non-English speaking countries. However, users need to communicate with the computer in their native language. When that native language involves thousands of characters, the 256 characters allowed by the EBCDIC set is clearly not sufficient.

To support languages with a large number of characters, VS Pascal supports a predefined scalar data type, GCHAR, that represents one DBCS character. VS Pascal also supports a predefined structured data type, GSTRING, that represents a DBCS string. Both input and output of DBCS data require terminals and printers with DBCS capability, which are available in countries that extensively use those languages.

DBCS constants and comments require the use of a shift-out (X'0E') and shift-in (X'0F') character. The shift-out character indicates the beginning of DBCS data; the shift-in character indicates the end of DBCS data. The GRAPHIC compile-time option must be active for VS Pascal to recognize the shift-out and shift-in characters. DBCS character constants and comments cannot span multiple lines.

Each byte of a DBCS character must be within the range X'41' through X'FE' inclusive. A DBCS blank is X'4040'.

For an example of DBCS constants, comments, GCHAR data type, and GSTRING data type, see the *VS Pascal Language Reference.*

Appendix G. Migration Considerations

From VS Pascal Release 1 to VS Pascal Release 2

VS Pascal Release 2 supports all existing Release 1 functions with full upward compatibility at both the source and object levels, with the exceptions noted in Figure 111.

Release 2 Change to ...	Message Issued	Nature of Change
Debugged Object Decks	—	Release 1 object decks compiled with DEBUG must be recompiled for use with Release 2. This allows the debugger to work with larger programs.
Compile-Time Limits	AMPX902S	In Release 2, routines compiled with the HEADER compile-time option, which places header information in the code space, might exceed 8K. To correct the error: • Compile routines with the NOHEADER compile-time option • Issue a %UHEADER OFF before the routine if a user header is being used.
Compiler Option Checking	Messages in the Range 7xx	Release 2 now flags some invalid compile-time options that were ignored in Release 1. This affects only those users who invoke the compiler with their own commands.
Compiler Directive Checking	—	Release 2 now flags some invalid compiler directives that were ignored in Release 1.
Hex String Checking	819	Release 2 flags hex string literals containing an odd number of hex digits (for example, '404'XC) as errors. Release 1 added a zero to the right of hex string literals that contained an odd number of hex digits.

Figure 111 (Part 1 of 4). Exceptions in VS Pascal Release 2 Support of Release 1

Release 2 Change to ...	Message Issued	Nature of Change
DBCS Checking	—	The GRAPHIC compile-time option now causes VS Pascal to check double-byte character set (DBCS) literals and comments and the %TITLE and %WRITE compiler directives to ensure that: • A shift-out ('OE'X) character and shift-in ('OF'X) character are paired before the end of a source record. (This is the only check done by Release 1.) • Every shift-in is preceded by a shift-out. • There are an even number of bytes between a shift-out and shift-in. • Only valid DBCS characters occur between a shift-out and shift-in.
Structured Constant Checking	—	As defined in the proposed ANSI/IEEE Extended Pascal Standard, Release 2 now flags as errors structured constants containing files.
VAR Parameter Checking	—	Release 2 now flags as errors actual and formal VAR parameters that do not have the same size.
Writing Character Data	—	As defined in the proposed ANSI/IEEE Extended Pascal Standard, when a character variable or constant is written with a field width of zero, no data will be written.
"=" Operator	—	Compile-time and run-time options no longer accept "=". In Release 2 you must specify optname = optvalue as optname(optvalue). **Note:** Only the ERRCOUNT, ERRFILE, STACK, and HEAP run-time options, and the LINECOUNT and PAGEWIDTH compile-time options, are affected.

Figure 111 (Part 2 of 4). Exceptions in VS Pascal Release 2 Support of Release 1

Release 2 Change to ...	Message Issued	Nature of Change
New "> <" Operator	—	When the characters "> <" are passed in a string to TOKEN or LTOKEN, they are now returned as one token rather than two, unless the ">" was returned as part of another token. **Note:** The characters "> <" can now be used as a set operator for symmetric difference as defined in the proposed ANSI/IEEE Extended Pascal Standard. As an IBM extension to the Standard, these characters can also be used as Boolean exclusive or. Release 2 still supports XOR and "&&" so that existing code need not be updated. However, new code should use only "> <".
MAIN and REENTRANT Routine Directives	—	Release 2 now flags as errors MAIN and REENTRANT routines that do not have their bodies declared.
ONERROR Routines	AMPX600 AMPX601 AMPX602 AMPX700 AMPX701 AMPX702 AMPX081 AMPX082 AMPX083 AMPX084	As part of national language support, ONERROR routines that caused message AMPX047 in Release 1 now cause messages AMPX600-602 in Release 2. Routines that caused message AMPX057 in Release 1 now cause messages AMPX700-702 in Release 2. Routines that caused message AMPX089 now cause message AMPX081 in Release 2. Routines that caused messages AMPX086 through AMPX088 in Release 1 now cause messages AMPX082-084 in Release 2.
LPAD and RPAD Procedures	—	You should delete any %INCLUDE STRING directives from your source code if you want to use LPAD and RPAD with DBCS data. **Note:** If you have declared LPAD or RPAD in a higher scope than the one in which the %INCLUDE STRING was deleted, you must use different names in order to be able to access the predefined LPAD and RPAD routines.
READSTR Procedure	—	When a fieldwidth of READSTR is equal to zero, the length of the string will be used as the fieldwidth; if the fieldwidth is less than zero, the absolute value of the fieldwidth will be used as the fieldwidth. This makes READSTR consistent with READ.

Figure 111 (Part 3 of 4). Exceptions in VS Pascal Release 2 Support of Release 1

Release 2 Change to ...	Message Issued	Nature of Change
PROLOG Macro in Assembler Routines	—	Any code using the PROLOG macro should be re-assembled. This allows assembler routine errors to generate an error trace back rather than error message AMPX902S, and improves error checking. The new parameter FPARMS should be added to the PROLOG macro of any assembler code that contains local variables as well as formal parameters. This helps the PROLOG macro generate code compatible with compiler-generated code, and might prevent certain memory errors.

Figure 111 (Part 4 of 4). Exceptions in VS Pascal Release 2 Support of Release 1

From Pascal/VS Release 2.2 to VS Pascal Release 1

VS Pascal Release 2 supports all functions that are in Pascal/VS Release 2.2 (the Program Offering that preceded the VS Pascal Release 1 Licensed Program) with full upward compatibility at the source level except for some minor enhancements. Recompilation of the Pascal/VS source code under VS Pascal Release 2 is required.

The main differences between Pascal/VS Release 2.2 and VS Pascal Release 1 are listed in the following section; the main differences between VS Pascal Release 1 and VS Pascal Release 2 are listed in the preceding section.

Changes In	Change
Source Language Statements	Threatened FOR loop indexes are now always flagged with a warning. Value parameters may no longer be control variables for FOR loops.

Figure 112 (Part 1 of 4). Exceptions to VS Pascal Release 1 Support of Pascal/VS Release 2.2

Changes in	Change
Compiler Options	The WARNING | NOWARNING compile-time options are no longer supported. They have been replaced by the FLAG compiler option. VS Pascal supports only two language level compile-time options: ANSI83 and EXTENDED. Pascal/VS supported three compile-time options: STANDARD, STDRES, and EXTENDED. When LANGLVL(ANSI83) is specified, all standard violations will be flagged as compiler errors. If warnings (instead of errors) are desired, STDFLAG(W) must be specified. This is equivalent to LANGLVL(STDRES). You cannot specify only the option, such as EXTENDED for LANGLVL. You must specify the complete option, such as LANGLVL(EXTENDED).

Run-Time Library Routines	The component types of the arrays passed to the PACK and UNPACK routines must have equal ranges if they are subrange data types.
	
	Range checking will now be done on the integer-to-character conversion function (CHR).
	
	In Pascal/VS, if the third parameter to SUBSTR or DELETE was a variable with the value −1, the remainder of the string was returned or deleted. In VS Pascal, an error message is issued instead. If the Pascal/VS behavior is required, a statement such as:
	
	`IF L = −1`
	`  THEN S := SUBSTR(S,I)`
	`  ELSE S := SUBSTR(S,I,L);`
	
	should produce results equivalent to those in Pascal/VS.

Function Declarations	Function results must now be identifier types. This will only prevent subrange specifications whose first value is a constant identifier. Each function must now contain an assignment to the function result.

Figure 112 (Part 2 of 4). Exceptions to VS Pascal Release 1 Support of Pascal/VS Release 2.2

Changes in	Change

I/O Routines	Parameters for the I/O routines READ, READLN, READSTR, WRITE, WRITELN, and WRITESTR are now evaluated in left-to-right order.
	
	Output of some real numbers will be changed to conform to the 1983 ANSI/IEEE standard. Pascal/VS always writes a real number with a value of zero as "0.0" or "0." The ANSI/IEEE standard makes no such distinction between zero and other real numbers. In VS Pascal, real zero is now handled just like any other real number. If the Pascal/VS behavior is required, a statement such as:
	
	`IF R=0.0`
	`  THEN WRITELN (R : LENGTH1 : 1)`
	
	should produce results equivalent to those in Pascal/VS.
	
	When writing REAL data: in the case where *TotalWidth* and *FracDigits* are specified and *FracDigits* equals zero, then a decimal point is written, but no decimal place is written. If *FracDigits* is negative, the number is written using the floating point form. If the Pascal/VS Release 2.2 behavior is required, a statement such as:
	
	`IF (LENGTH2 = 0) AND (ABS(R) >= 1)`
	`   THEN WRITELN(ROUND(R) : LENGTH1)`
	`   ELSE WRITELN(R : LENGTH1 : LENGTH2);`
	
	will replace WRITELN(R : LENGTH1 : LENGTH2).
	
	The CLOSE procedure no longer accepts open options.

Program Parameters	Duplicate program parameters are now flagged. Program parameters (other than INPUT or OUTPUT) that are not declared as global variables are now flagged. If INPUT is specified as a program parameter, a RESET(INPUT) will be issued. If this behavior is not desired, remove INPUT from the program parameter list. If OUTPUT is specified as a program parameter, a REWRITE(OUTPUT) will be issued. If this behavior is not desired, remove OUTPUT from the program parameter list.

Figure 112 (Part 3 of 4). Exceptions to VS Pascal Release 1 Support of Pascal/VS Release 2.2

Changes In	Change
Operating Systems	The OS/VS 1 and VM/PC operating systems are not supported. Although VS Pascal and its generated programs may work on these systems, any problems on these systems must be reproduced on a supported operating system. The IBM Support Center will not accept problem reports using these operating systems.
Routine Parameters	Variables preceded by a unary plus and variables in parentheses are no longer allowed as actual VAR parameters. Range checking is now done on all pass-by-value and pass-by-constant actual expression parameters.
Other Items	Illegal use of file variables (embedded files) not flagged by Pascal/VS will now be diagnosed by VS Pascal. There were situations in which file variables occurred illegally, and Pascal/VS did not flag such occurrences as illegal. Illegal occurrences of file variables include: files within files, assigning files, files in value parameters, and files in function results. Fields in records may no longer have the same name as the domain type of a new pointer type being referenced in the record. Invalid values for value assignments are now flagged as errors. Global labels in segment units are now flagged with a warning message because they can't be branched to. All tag constants in a variant record must be legal values for the tag type of the record. If you do not use the %INCLUDE ONERROR method to define ONERROR, you must change the ALPHA parameters to CONST conformant STRING parameters.

Figure 112 (Part 4 of 4). Exceptions to VS Pascal Release 1 Support of Pascal/VS Release 2.2

Glossary

This glossary defines terms and acronyms used in this book. It includes terms and definitions from the IBM *Dictionary of Computing*, SC20-1699.

A

addressing mode. Determines whether generated instructions use 24-bit or 31-bit addressing.

AMODE. Specifies the addressing mode (of an entry point in a load module) in effect when the load module is entered at run-time.

ANSI/IEEE standard. American National Standards Institute 1983 standard for the Pascal Computer Programming Language.

B

BDAM (basic direct access method). An access method used to directly retrieve or update particular blocks of a data set on a direct access device.

bit. One binary digit.

BPAM (basic partitioned access method). An access method that can be applied to create program libraries in direct access storage for convenient storage and retrieval of programs.

byte. The unit of addressability on the System/370; its length is 8 bits.

C

cataloged procedure. A regularly-used set of job control statements that are prepared once, given a name, stored in a system library, and the name entered in the catalog for that library. A cataloged procedure comprises one or more job steps (though it is not a job because it must not contain a JOB statement).

compilable unit. An independently compilable piece of code. There are two types of unit: the program unit and the segment unit.

compile-time option. A parameter passed to the compiler that specifies whether or not a particular feature is to be active.

constant. A value that is either a literal or an identifier that has been associated with a value in a CONST declaration.

constant expression. An expression that can be completely evaluated by the compiler at compile time.

current heap. The area of storage in the VS Pascal run-time environment where dynamic variables allocated by calls to NEW will reside. While many heaps can exist at one time, there is only one current heap.

D

DBCS (double-byte character set). The internal representation of each character requiring 2 bytes of space. Languages such as Kanji require such double-byte representations.

DCB (data control block). A control block used by access method routines in storing and retrieving data.

DD statement (data definition statement). The JCL statement that identifies the input/output facilities required by the program executed in the job step.

delimiter statement. The JCL statement that separates data in the input stream from the job control statements that follow this data.

dynamic variable. A variable that is allocated under programmer control. Explicit allocates and deallocates are required; the predefined procedures NEW and DISPOSE are provided for this purpose.

E

EBCDIC. See extended binary-coded decimal interchange code.

element. The component of an array.

enumerated scalar type. A scalar that is defined by enumerating the elements of the type. Each element is represented by an identifier.

EXEC statement. The JCL statement that identifies a job step and the program to be executed, either directly or by means of a cataloged procedure.

executable program. Consists of object code from your main program that is link-edited with the object code from the run-time library and any segments that are needed by the main program.

extended binary-coded decimal interchange code (EBCDIC). The underlying character set used in VS Pascal.

external routine. A procedure or function whose body is not contained in the unit being compiled.

external variable. A variable that may be referenced from units and scopes other than the one in which it was declared.

F

field. The component of a record.

file pointer. A pointer into an input/output buffer.

FILE type. A data type that is the mechanism to do input and output in VS Pascal.

fixed-length record. A record having the same length as all other records with which it is logically or physically associated.

fixed string. A PACKED ARRAY[1...n] OF CHAR or a PACKED ARRAY[1...n] OF GCHAR.

floating-point number. A subset of the set of real numbers.

formal parameter. A parameter as declared in the routine heading. A formal parameter is used to specify what is permitted to be passed to a routine.

function. A routine that is invoked by coding its name in an expression. The routine passes a result back to the invoker through the routine name.

G

GENERIC routine. A routine whose parameter count, types and passing mechanism are specified by the call to the routine.

H

heap. An area of storage where dynamic variables are created.

hexadecimal digits. A digit that is a member of the set of sixteen digits: 0 through 9, and then A through F used in a number system of Base 16.

I

identifier. The name of a declared item.

integer. The set of positive and negative whole numbers.

internal routine. A routine that can be used only from within the lexical scope in which it was declared.

J

JCL (job control language). A control language used to identify a job to an operating system and to describe the job's requirements.

JOB statement. The JCL statement that identifies the start of the job.

L

lexical scope. Identifies the portion of a unit in which a name is known. An identifier declared in a routine is known within that routine and within all nested routines. If a nested routine declares an item with the same name, the outer item is not available in the nested routine.

load module. A computer program in a form suitable for loading into main storage for execution.

M

mixed string. A mixture of DBCS and SBCS data.

O

object code. Output from a compiler or assembler which is itself executable machine code or is suitable for processing to produce executable machine code.

object module. A portion of an object program suitable as input to a linkage editor.

ordinal type. A scalar type whose values are mapped to a continuous range of integers.

P

PDS (partitioned data set). A file (sometimes referred to as a library) containing logical files that are called members.

pointer. A variable that contains the address of a dynamic variable.

procedure. A routine, invoked by coding its name as a statement, that does not pass a result back to the invoker.

program. See executable program.

program unit. The name of the compilable unit of code that represents the first unit executed.

Q

QSAM (queued sequential access method). An extended version of BSAM. When this method is used, a queue is formed of input data blocks that are awaiting processing or of output data blocks that have been processed and are awaiting transfer to auxiliary storage or to an output device.

qualification. A naming mechanism in a programming language for referencing a component of a language

object by means of a reference to the object and an identifier declared for the component. A qualification consists of a unit name and a routine name.

R

record type. The structured type that contains a series of fields. Each field may be a different type than the other fields in the record. A field is selected by the name of the field.

register. A storage device having a specified storage capacity such as a bit, byte, or computer word, and usually intended for a special purpose.

reserved word. An identifier whose use is restricted by the VS Pascal compiler.

residence mode. Determines where in storage a program may reside (either above or below the 16-megabyte line).

RMODE. Specifies the residence mode of a load module when it is loaded into virtual storage for execution.

routine. A unit of a VS Pascal program that may be called. The two types of routines are functions and procedures.

S

scalar type. A type whose values contain only one element.

SBCS. See single-byte character set.

segment unit. A compilable unit in VS Pascal that is used to contain entry routines.

shift-in character. Indicates the end of DBCS data and is denoted by X

source module. The source statements that constitute the input to a language translator for a particular translation.

statement. The executable code in a VS Pascal program.

string. Represents an ordered list of characters whose size may vary at run time. There is a maximum size for every string.

string constant. A string whose value is fixed by the compiler.

structured type. Any one of several data type mechanisms that defines variables that have multiple values. Each value is referred to generally as a component.

subrange type. Used to define a variable whose value is restricted to some subset of values of a base ordinal type.

subheap. An area in a heap delimited by a call to MARK. Subheaps are treated in a stack-like manner within a heap.

T

text decks. A main program unit and any segment units of file type TEXT.

text libraries. The VS Pascal run-time library and any other libraries of compiled code with the file type TXTLIB.

type. Defines the permissible values a variable may assume.

type definition. The specification of a data type. The specification may appear in a type declaration or in the declaration of a variable.

type identifier. The name given to a declared type.

U

unit. See compilable unit.

V

variable-length record. A record having a length independent of the length of other records with which it is logically or physically associated.

Bibliography

VS Pascal Publications

These books provide additional information about VS Pascal.

Evaluation

- *VS Pascal General Information*, GC26-4318, provides an overview of VS Pascal.
- *VS Pascal Licensed Program Specifications*, GC26-4317, contains warranty information for VS Pascal.

Application Programming

- *VS Pascal Language Reference*, SC26-4320, provides a detailed explanation of the VS Pascal programming language and its syntax.
- *VS Pascal Reference Summary*, SX26-3760, provides quick-reference charts of VS Pascal language rules and processing/debugging options.

Installation

- *VS Pascal Installation and Customization for VM*, SC26-4342, explains how to install VS Pascal under VM/SP and VM/XA.
- *VS Pascal Installation and Customization for MVS*, SC26-4321, explains how to install VS Pascal under MVS/SP, MVS/XA and MVS/ESA.

Diagnosis

- *VS Pascal Diagnosis Guide and Reference*, LY27-9525, explains how to diagnose, report, and request information on VS Pascal-related problems.

Related Publications

These publications will be helpful when working with VS Pascal.

VM

Virtual Machine/System Product CMS Command Reference, SC19-6209

Virtual Machine/System Product CMS User's Guide, SC19-6210

Virtual Machine/System Product: Planning Guide and Reference, SC19-6201

Virtual Machine/System Product CMS for System Programming, SC24-5286

Virtual Machine/System Product Application Development Guide, SC24-5247

VM/XA System Product CMS Application Program Development Guide, SC23-0355

VM/XA System Product CMS Application Program Conversion Guide, SC23-0403

VM/XA System Product CMS Command Reference, SC23-0354

VM/XA System Product CMS User's Guide, SC23-0356

VM/XA System Product Planning, GC23-0378

Job Control Language

MVS/370 JCL User's Guide, GC28-0692

MVS/370 JCL Reference, GC28-1350

MVS/Extended Architecture JCL User's Guide, GC28-1351

MVS/Extended Architecture JCL Reference, GC28-1352

MVS/Enterprise Systems Architecture JCL User's Guide, GC28-1830.

MVS/Enterprise Systems Architecture JCL Reference, GC28-1829.

TSO

TSO Extensions Command Reference, GC28-1307

OS/VS2 TSO Command Language Reference, GC28-0646

MVS/Extended Architecture TSO Command Language Reference, GC28-0646 with supplement number GD23-0259

Assembler

Assembler H Version 2 Application Programming: Language Reference, GC26-4037.

COBOL

IBM OS/VS COBOL Compiler and Library Programmer's Guide, SC28-6483

VS COBOL II Application Programming Guide, SC26-4045

PL/I

OS PL/I Optimizing Compiler: Programmer's Guide, SC33-0006

OS PL/I Version 2 Programming Guide, SC26-4307.

OS PL/I Version 2 Programming: Language Reference, SC26-4308.

FORTRAN

VS FORTRAN Programming Guide, SC26-4118

VS FORTRAN Version 2: Programming Guide, SC26-4222

Linkage Editor and Loader

MVS/370 Linkage Editor and Loader User's Guide, GC26-4061

MVS/Extended Architecture Linkage Editor and Loader User's Guide (Data Facility Product, Version 1), GC26-4011

MVS/Extended Architecture Linkage Editor and Loader User's Guide (Data Facility Product, Version 2), GC26-4143

Principles of Operation

IBM System/370 Extended Architecture Principles of Operation, SA22-7085

IBM System/370 Principles of Operation, GA22-7000

IBM System/370 Enterprise Systems Architecture Principles of Operation, SA22-7200.

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Numerics

VS Pascal
Application Programming Guide

SC26-4319-1

Reader's Comment Form

This manual is part of a library that serves as a reference source for system analysts, programmers, and operators of IBM systems. 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: Do not use this form to request IBM publications. If you do, your order will be delayed because publications are not stocked at the address printed on the reverse side. Instead, you should 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.

If you have applied any technical newsletters (TNLs) to this book, please list them here: _____

Comments (please include specific chapter and page references) :

Note: Staples can cause problems with automatic mail-sorting equipment.
Please use pressure-sensitive or other gummed tape to seal this form.

If you want a reply, please complete the following information:

Name _____ Date _____

Company _____ Phone No. (_____) _____

Address _____

_____

Thank you for your cooperation. No postage is 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 them directly to the address in the Edition Notice on the back of the title page.)

SC26-4319-1

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 MAIL PERMIT NO. 40 ARMONK, NY

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Programming Publishing
P.O. Box 49023
San Jose, CA 95161-9023

Fold and tape | Please do not staple | Fold and tape

IBM®

IBM®

Program Number
5668-767
5668-717

File Number
S370-40

The VS Pascal Library

Application Programming Guide	SC26-4319
Diagnosis Guide and Reference	LY27-9525
General Information	GC26-4318
Installation and Customization for MVS	SC26-4321
Installation and Customization for VM	SC26-4342
Language Reference	SC26-4230

Supplementary Publications

Licensed Program Specifications	CG264317
Reference Summary	SX26-3760

SC26-4319-1

Printed in U.S.A.